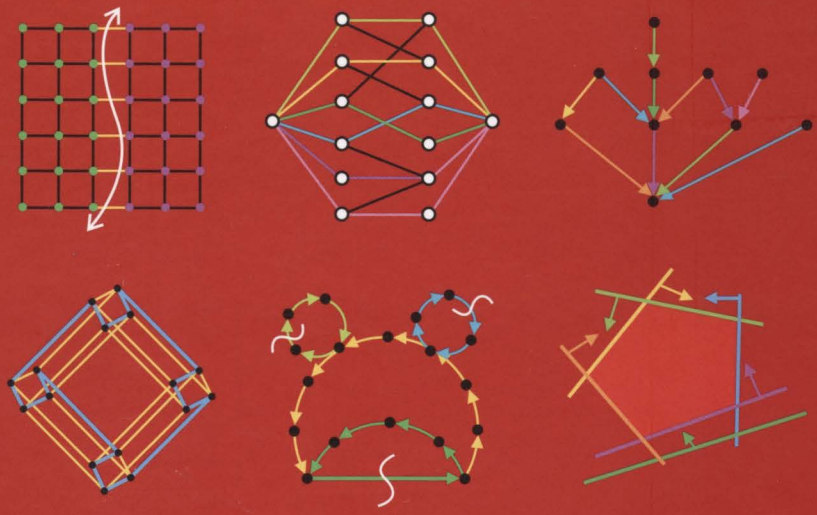
Третье издание

# Алгоритмы

## Руководство по разработке



Стивен Скиена

Springer

bhv®

+ Материалы на www.bhv.ru

Steven S. Skiena

# The Algorithm Design Manual

Third Edition

Springer

Стивен С. Скиена

# Алгоритмы
## Руководство по разработке

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 681.3.06
ББК 32.973.26-018.2
С42

**Скиена С. С.**

С42 Алгоритмы. Руководство по разработке. — 3-е изд.: Пер. с англ. — СПб.: БХВ-Петербург, 2022. — 848 с.: ил.

ISBN 978-5-9775-6799-2

Книга является наиболее полным руководством по разработке эффективных алгоритмов. Первая часть книги содержит практические рекомендации по разработке алгоритмов: приводятся основные понятия, дается анализ алгоритмов, рассматриваются типы структур данных, основные алгоритмы сортировки, операции обхода графов и алгоритмы для работы со взвешенными графами, примеры использования комбинаторного поиска, эвристических методов и динамического программирования. Вторая часть книги содержит обширный список литературы и каталог из 75 наиболее распространенных алгоритмических задач, для которых перечислены существующие программные реализации.

В третьем издании расширен набор рандомизированных алгоритмов, алгоритмов хеширования, аппроксимации и квантовых вычислений. Добавлено более 100 новых задач, даны ссылки к реализациям на C, C++ и Java.

Книгу можно использовать в качестве справочника по алгоритмам для программистов, исследователей и в качестве учебного пособия для студентов соответствующих специальностей.

УДК 681.3.06
ББК 32.973.26-018.2

First published in English under the title
The Algorithm Design Manual
by Steven S. Skiena, edition: 3
Copyright © The Editor(s) (if applicable) and The Author(s), under exclusive license to
Springer Nature Switzerland AG, 2020
This edition has been translated and published under licence from
Springer Nature Switzerland AG.
Springer Nature Switzerland AG takes no responsibility and shall not be made liable for the accuracy of the translation.

Впервые опубликовано на английском языке под названием
The Algorithm Design Manual
Steven S. Skiena, edition: 3
© Springer Nature Switzerland AG, 2020
Издание переведено и опубликовано по лицензии Springer Nature Switzerland AG.
Springer Nature Switzerland AG не несет ответственности за точность перевода.

ISBN 978-3-030-54255-9 (англ.)
ISBN 978-5-9775-6799-2 (рус.)

© Springer Nature Switzerland AG, 2020
© Перевод на русский язык, оформление. ООО "БХВ-Петербург", ООО "БХВ", 2022

# Оглавление

**Предисловие** ..... **17**
Читателю ..... 17
Преподавателю ..... 19
Благодарности ..... 20
Ограничение ответственности ..... 21

**ЧАСТЬ I. ПРАКТИЧЕСКАЯ РАЗРАБОТКА АЛГОРИТМОВ** ..... **23**

**Глава 1. Введение в разработку алгоритмов** ..... **25**
1.1. Оптимизация маршрута робота ..... 26
1.2. Задача календарного планирования ..... 30
1.3. Обоснование правильности алгоритмов ..... 33
1.3.1. Задачи и свойства ..... 34
1.3.2. Представление алгоритмов ..... 35
1.3.3. Демонстрация неправильности алгоритма ..... 35
ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: «Жадные» кинозвезды? ..... 37
1.3.4. Индукция и рекурсия ..... 38
ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Правильность инкрементных алгоритмов ..... 39
1.4. Моделирование задачи ..... 40
1.4.1. Комбинаторные объекты ..... 41
1.4.2. Рекурсивные объекты ..... 42
1.5. Доказательство от противного ..... 44
1.6. Истории из жизни ..... 45
1.7. История из жизни. Моделирование проблемы ясновидения ..... 45
1.8. Прикидка ..... 49
Замечания к главе ..... 50
1.9. Упражнения ..... 50
Поиск контрпримеров ..... 50
Доказательство правильности ..... 51
Математическая индукция ..... 52
Приблизительные подсчеты ..... 52
Проекты по реализации ..... 53
Задачи, предлагаемые на собеседовании ..... 53
LeetCode ..... 54
HackerRank ..... 54
Задачи по программированию ..... 54

**Глава 2. Анализ алгоритмов ..... 55**
2.1. Модель вычислений RAM ..... 55
2.1.1. Анализ сложности наилучшего, наихудшего и среднего случая ..... 56
2.2. Асимптотические («Big Oh») обозначения ..... 58
Остановка для размышлений: Возвращение к определениям ..... 61
Остановка для размышлений: Квадраты ..... 61
2.3. Скорость роста и отношения доминирования ..... 61
2.3.1. Отношения доминирования ..... 63
2.4. Работа с асимптотическими обозначениями ..... 64
2.4.1. Сложение функций ..... 64
2.4.2. Умножение функций ..... 64
Остановка для размышлений: Транзитивность ..... 65
2.5. Оценка эффективности ..... 65
2.5.1. Сортировка методом выбора ..... 65
Доказываем временную сложность Θ-большое ..... 66
2.5.2. Сортировка вставками ..... 67
Доказываем временную сложность Θ-большое ..... 68
2.5.3. Сравнение строк ..... 68
Доказываем время исполнения по Θ-большое ..... 69
2.5.4. Умножение матриц ..... 70
2.6. Суммирование ..... 71
Остановка для размышлений: Формулы факториала ..... 72
2.7. Логарифмы и их применение ..... 73
2.7.1. Логарифмы и двоичный поиск ..... 73
2.7.2. Логарифмы и деревья ..... 74
2.7.3. Логарифмы и биты ..... 74
2.7.4. Логарифмы и умножение ..... 75
2.7.5. Быстрое возведение в степень ..... 75
2.7.6. Логарифмы и сложение ..... 76
2.7.7. Логарифмы и система уголовного судопроизводства ..... 76
2.8. Свойства логарифмов ..... 77
Остановка для размышлений: Важно ли деление точно пополам ..... 79
2.9. История из жизни. Загадка пирамид ..... 79
2.10. Анализ высшего уровня (*) ..... 82
2.10.1. Малораспространенные функции ..... 83
2.10.2. Пределы и отношения доминирования ..... 84
Замечания к главе ..... 85
2.11. Упражнения ..... 86
Анализ программ ..... 86
Упражнения по асимптотическим обозначениям ..... 87
Суммирование ..... 92
Логарифмы ..... 93
Задачи, предлагаемые на собеседовании ..... 93
LeetCode ..... 94
HackerRank ..... 94
Задачи по программированию ..... 94

**Глава 3. Структуры данных ... 95**
3.1. Смежные и связные структуры данных ... 95
3.1.1. Массивы ... 96
3.1.2. Указатели и связные структуры данных ... 97
Поиск элемента в связном списке ... 98
Вставка элемента в связный список ... 99
Удаление элемента из связного списка ... 99
3.1.3. Сравнение ... 100
3.2. Стеки и очереди ... 101
3.3. Словари ... 102
Остановка для размышлений: Сравнение реализаций словаря (I) ... 103
Остановка для размышлений: Сравнение реализаций словаря (II) ... 105
3.4. Двоичные деревья поиска ... 108
3.4.1. Реализация двоичных деревьев ... 108
Поиск в дереве ... 109
Поиск наименьшего и наибольшего элементов дерева ... 110
Обход дерева ... 110
Вставка элементов в дерево ... 111
Удаление элемента из дерева ... 112
3.4.2. Эффективность двоичных деревьев поиска ... 112
3.4.3. Сбалансированные деревья поиска ... 113
Остановка для размышлений: Использование сбалансированных деревьев поиска ... 114
3.5. Очереди с приоритетами ... 115
Остановка для размышлений: Построение базовых очередей с приоритетами ... 115
3.6. История из жизни. Триангуляция ... 117
3.7. Хеширование ... 121
3.7.1. Коллизии ... 121
3.7.2. Выявление дубликатов с помощью хеширования ... 123
3.7.3. Прочие приемы хеширования ... 125
3.7.4. Каноникализация ... 125
3.7.5. Уплотнение ... 126
3.8. Специализированные структуры данных ... 126
3.9. История из жизни. Геном человека ... 127
Замечания к главе ... 131
3.10. Упражнения ... 131
Стеки, очереди и списки ... 131
Элементарные структуры данных ... 132
Деревья и другие словарные структуры ... 132
Применение древовидных структур ... 134
Задачи по реализации ... 135
Задачи, предлагаемые на собеседовании ... 136
LeetCode ... 137
HackerRank ... 137
Задачи по программированию ... 137

**Глава 4. Сортировка и поиск ... 138**
4.1. Применение сортировки ... 138
Остановка для размышлений: Поиск пересечения множеств ... 141
Остановка для размышлений: Использование хеша для решения задач ... 142

4.2. Практические аспекты сортировки ..... 143
4.3. Пирамидальная сортировка ..... 145
4.3.1. Пирамиды ..... 146
Остановка для размышлений: Поиск в пирамиде ..... 148
4.3.2. Создание пирамиды ..... 148
4.3.3. Наименьший элемент пирамиды ..... 149
4.3.4. Быстрый способ создания пирамиды (*) ..... 151
Остановка для размышлений: Расположение элемента в пирамиде ..... 153
4.3.5. Сортировка вставками ..... 154
4.4. История из жизни. Билет на самолет ..... 155
4.5. Сортировка слиянием. Метод «разделяй и властвуй» ..... 158
4.6. Быстрая сортировка. Рандомизированная версия ..... 160
4.6.1. Интуиция: ожидаемое время исполнения алгоритма быстрой сортировки ..... 163
4.6.2. Рандомизированные алгоритмы ..... 164
Остановка для размышлений: Болты и гайки ..... 166
4.6.3. Действительно ли алгоритм быстрой сортировки работает быстро? ..... 167
4.7. Сортировка распределением. Метод блочной сортировки ..... 167
4.7.1. Нижние пределы для сортировки ..... 168
4.8. История из жизни. Скиена в суде ..... 170
Замечания к главе ..... 172
4.9. Упражнения ..... 172
Применение сортировки: сортировка чисел ..... 172
Применение сортировки: интервалы и множества ..... 174
Пирамиды ..... 175
Быстрая сортировка ..... 175
Сортировка слиянием ..... 176
Другие алгоритмы сортировки ..... 177
Нижние пределы ..... 178
Поиск ..... 178
Задачи по реализации ..... 178
Задачи, предлагаемые на собеседовании ..... 179
LeetCode ..... 179
HackerRank ..... 180
Задачи по программированию ..... 180

**Глава 5. Метод «разделяй и властвуй» ..... 181**
5.1. Двоичный поиск и связанные с ним алгоритмы ..... 181
5.1.1. Частота вхождения элемента ..... 182
5.1.2. Односторонний двоичный поиск ..... 183
5.1.3. Корни числа ..... 184
5.2. История из жизни. Поиск «бага в баге» ..... 184
5.3. Рекуррентные соотношения ..... 186
5.3.1. Рекуррентные соотношения метода «разделяй и властвуй» ..... 187
5.4. Решение рекуррентных соотношений типа «разделяй и властвуй» (*) ..... 188
5.5. Быстрое умножение ..... 190
5.6. Поиск наибольшего поддиапазона и ближайшей пары ..... 192
5.7. Параллельные алгоритмы ..... 194
5.7.1. Параллелизм на уровне данных ..... 194
5.7.2. Подводные камни параллелизма ..... 195

5.8. История из жизни. «Торопиться в никуда» ... 196
5.9. Свертка (*) ... 197
5.9.1. Применение свертки ... 198
5.9.2. Быстрое полиномиальное умножение (**) ... 199
Замечания к главе ... 202
5.10. Упражнения ... 202
Двоичный поиск ... 202
Алгоритмы типа «разделяй и властвуй» ... 203
Рекуррентные соотношения ... 203
LeetCode ... 204
HackerRank ... 204
Задачи по программированию ... 205

**Глава 6. Хеширование и рандомизированные алгоритмы ... 206**
Остановка для размышлений: Город быстрой сортировки ... 207
6.1. Обзор теории вероятностей ... 207
6.1.1. Теория вероятностей ... 207
6.1.2. Составные события и независимость ... 209
6.1.3. Условная вероятность ... 210
6.1.4. Распределения вероятностей ... 211
6.1.5. Среднее и дисперсия ... 212
6.1.6. Броски монет ... 212
Остановка для размышлений: Случайный обход графа ... 213
6.2. Задача мячиков и контейнеров ... 214
6.2.1. Задача о собирании купонов ... 216
Остановка для размышлений: Время покрытия для $K_n$ ... 216
6.3. Почему хеширование является рандомизированным алгоритмом? ... 217
6.4. Фильтры Блума ... 218
6.5. Парадокс дня рождения и идеальное хеширование ... 220
6.6. Метод минимальных хеш-кодов ... 222
Остановка для размышлений: Приблизительная оценка численности популяции ... 224
6.7. Эффективный поиск подстроки в строке ... 225
6.8. Тестирование чисел на простоту ... 226
6.9. История из жизни. Как я дал Кнуту свой средний инициал ... 228
6.10. Откуда берутся случайные числа? ... 229
Замечания к главе ... 230
6.11. Упражнения ... 230
Вероятность ... 230
Хеширование ... 231
Рандомизированные алгоритмы ... 231
LeetCode ... 232
HackerRank ... 232
Задачи по программированию ... 232

**Глава 7. Обход графов ... 233**
7.1. Разновидности графов ... 234
7.1.1. Граф дружеских отношений ... 237
7.2. Структуры данных для графов ... 240
7.3. История из жизни. Жертва закона Мура ... 244

7.4. История из жизни. Создание графа ..... 248
7.5. Обход графа ..... 250
7.6. Обход в ширину ..... 251
7.6.1. Применение обхода ..... 254
7.6.2. Поиск путей ..... 255
7.7. Применение обхода в ширину ..... 256
7.7.1. Компоненты связности ..... 256
7.7.2. Раскраска графов двумя цветами ..... 257
7.8. Обход в глубину ..... 259
7.9. Применение обхода в глубину ..... 263
7.9.1. Поиск циклов ..... 263
7.9.2. Шарниры графа ..... 264
7.10. Обход в глубину ориентированных графов ..... 268
7.10.1. Топологическая сортировка ..... 270
7.10.2. Сильно связные компоненты ..... 272
Замечания к главе ..... 274
7.11. Упражнения ..... 275
Алгоритмы для эмуляции графов ..... 275
Обход графов ..... 276
Приложения ..... 277
Разработка алгоритмов ..... 278
Ориентированные графы ..... 280
Шарниры графа ..... 281
Задачи, предлагаемые на собеседовании ..... 282
LeetCode ..... 282
HackerRank ..... 282
Задачи по программированию ..... 282

**Глава 8. Алгоритмы для работы со взвешенными графами ..... 283**
8.1. Минимальные остовные деревья ..... 284
8.1.1. Алгоритм Прима ..... 285
8.1.2. Алгоритм Крускала ..... 288
8.1.3. Структура данных непересекающихся множеств ..... 290
8.1.4. Разновидности остовных деревьев ..... 293
8.2. История из жизни. И всё на свете — только сети ..... 295
8.3. Поиск кратчайшего пути ..... 298
8.3.1. Алгоритм Дейкстры ..... 299
Остановка для размышлений: Кратчайший путь с учетом веса вершин ..... 302
8.3.2. Кратчайшие пути между всеми парами вершин ..... 302
8.3.3. Транзитивное замыкание ..... 304
8.4. История из жизни. Печатаем с помощью номеронабирателя ..... 305
8.5. Потоки в сетях и паросочетание в двудольных графах ..... 309
8.5.1. Паросочетание в двудольном графе ..... 310
8.5.2. Вычисление потоков в сети ..... 311
8.6. Произвольный минимальный разрез ..... 315
8.7. Разрабатывайте не алгоритмы, а графы ..... 316
Остановка для размышлений: Нить Ариадны ..... 317
Остановка для размышлений: Упорядочивание последовательности ..... 317

Остановка для размышлений: Размещение прямоугольников по корзинам ....318
Остановка для размышлений: Конфликт имен файлов ....318
Остановка для размышлений: Разделение текста ....318
Замечания к главе ....319
8.8. Упражнения ....319
Алгоритмы для эмуляции графов ....319
Минимальные остовные деревья ....319
Поиск-объединение ....321
Поиск кратчайшего пути ....321
Потоки в сети и паросочетание ....323
LeetCode ....323
HackerRank ....323
Задачи по программированию ....323

**Глава 9. Комбинаторный поиск ....324**
9.1. Перебор с возвратом ....324
9.2. Примеры перебора с возвратом ....327
9.2.1. Генерирование всех подмножеств ....327
9.2.2. Генерирование всех перестановок ....329
9.2.3. Генерирование всех путей в графе ....330
9.3. Отсечение вариантов поиска ....332
9.4. Судоку ....333
9.5. История из жизни. Покрытие шахматной доски ....339
9.6. Поиск методом «лучший-первый» ....342
9.7. Эвристический алгоритм A* ....345
Замечания к главе ....347
9.8. Упражнения ....347
Перестановки ....347
Перебор с возвратом ....348
Игры и головоломки ....349
Комбинаторная оптимизация ....350
Задачи, предлагаемые на собеседовании ....350
LeetCode ....351
HackerRank ....351
Задачи по программированию ....351

**Глава 10. Динамическое программирование ....352**
10.1. Кэширование и вычисления ....353
10.1.1. Генерирование чисел Фибоначчи методом рекурсии ....353
10.1.2. Генерирование чисел Фибоначчи посредством кэширования ....354
10.1.3. Генерирование чисел Фибоначчи посредством динамического программирования ....356
10.1.4. Биномиальные коэффициенты ....358
10.2. Поиск приблизительно совпадающих строк ....360
10.2.1. Применение рекурсии для вычисления расстояния редактирования ....361
10.2.2. Применение динамического программирования для вычисления расстояния редактирования ....362
10.2.3. Восстановление пути ....364
10.2.4. Разновидности расстояния редактирования ....366

10.3. Самая длинная возрастающая подпоследовательность ..... 370
10.4. История из жизни. Сжатие текста для штрихкодов ..... 372
10.5. Неупорядоченное разбиение или сумма подмножества ..... 376
10.6. История из жизни: Баланс мощностей ..... 378
10.7. Задача упорядоченного разбиения ..... 381
10.8. Синтаксический разбор ..... 384
Остановка для размышлений: Экономичный синтаксический разбор ..... 386
10.9. Ограничения динамического программирования: задача коммивояжера ..... 387
10.9.1. Вопрос правильности алгоритмов динамического программирования ..... 388
10.9.2. Эффективность алгоритмов динамического программирования ..... 389
10.10. История из жизни. Динамическое программирование и язык Prolog ..... 390
Замечания к главе ..... 393
10.11. Упражнения ..... 394
Простые рекуррентные соотношения ..... 394
Расстояние редактирования ..... 394
«Жадные» алгоритмы ..... 396
Числовые задачи ..... 397
Задачи на графы ..... 399
Задачи по разработке ..... 399
Задачи, предлагаемые на собеседовании ..... 402
LeetCode ..... 402
HackerRank ..... 402
Задачи по программированию ..... 402

**Глава 11. NP-полнота ..... 403**
11.1. Сведение задач ..... 403
11.1.1. Ключевая идея ..... 404
11.1.2. Задачи разрешимости ..... 405
11.2. Сведение для создания новых алгоритмов ..... 406
11.2.1. Поиск ближайшей пары ..... 406
11.2.2. Максимальная возрастающая подпоследовательность ..... 407
11.2.3. Наименьшее общее кратное ..... 408
11.2.4. Выпуклая оболочка (*) ..... 409
11.3. Простые примеры сведения сложных задач ..... 410
11.3.1. Гамильтонов цикл ..... 410
11.3.2. Независимое множество и вершинное покрытие ..... 412
Остановка для размышлений: Сложность общей задачи календарного планирования ..... 413
11.3.3. Задача о клике ..... 415
11.4. Задача выполнимости булевых формул ..... 416
11.4.1. Задача выполнимости в 3-конъюнктивной нормальной форме (задача 3-SAT) ..... 417
11.5. Нестандартные сведения задачи SAT ..... 419
11.5.1. Вершинное покрытие ..... 419
11.5.2. Целочисленное программирование ..... 421
11.6. Искусство доказательства сложности ..... 423
11.7. История из жизни. Наперегонки со временем ..... 425
11.8. История из жизни. Полный провал ..... 427

11.9. Сравнение классов сложности P и NP ....430
11.9.1. Верификация решения и поиск решения ....430
11.9.2. Классы сложности P и NP ....431
11.9.3. Почему задача выполнимости является сложной? ....432
11.9.4. NP-сложность по сравнению с NP-полнотой ....432
Замечания к главе ....433
11.10. Упражнения ....434
Преобразования и выполнимость ....434
Базовые сведения ....435
Нестандартные сведения ....437
Алгоритмы для решения частных случаев задач ....438
P или NP? ....439
LeetCode ....439
HackerRank ....439
Задачи по программированию ....439

**Глава 12. Решение сложных задач ....440**
12.1. Аппроксимирующие алгоритмы ....440
12.2. Аппроксимация вершинного покрытия ....441
Остановка для размышлений: Вершинное покрытие в остатке ....443
12.2.1. Рандомизированный эвристический алгоритм вершинного покрытия ....444
12.3. Задача коммивояжера в евклидовом пространстве ....445
12.3.1. Эвристический алгоритм Кристофидеса ....446
12.4. Когда среднее достаточно хорошее ....448
12.4.1. Задача максимальной *k*-SAT ....448
12.4.2. Максимальный бесконтурный подграф ....449
12.5. Задача о покрытии множества ....449
12.6. Эвристические методы поиска ....451
12.6.1. Произвольная выборка ....452
Остановка для размышлений: Выбор пары ....454
12.6.2. Локальный поиск ....455
12.6.3. Имитация отжига ....459
Реализация ....462
12.6.4. Применение метода имитации отжига ....463
Задача максимального разреза ....463
Независимое множество ....463
Размещение компонентов на печатной плате ....464
12.7. История из жизни. Только это не радио ....465
12.8. История из жизни. Отжиг массивов ....468
12.9. Генетические алгоритмы и другие эвристические методы поиска ....471
12.9.1. Генетические алгоритмы ....471
12.10. Квантовые вычисления ....472
12.10.1. Свойства «квантовых» компьютеров ....473
12.10.2. Алгоритм Гровера для поиска в базе данных ....475
Решение задачи о выполнимости ....475
12.10.3. Более быстрое «преобразование Фурье» ....476
12.10.4. Алгоритм Шора для разложения целых чисел на множители ....477
12.10.5. Перспективы квантовых вычислений ....479
Замечания к главе ....481

12.11. Упражнения ..... 482
Частные случаи сложных задач ..... 482
Аппроксимирующие алгоритмы ..... 482
Комбинаторная оптимизация ..... 483
«Квантовые» вычисления ..... 484
LeetCode ..... 484
HackerRank ..... 484
Задачи по программированию ..... 484

**Глава 13. Как разрабатывать алгоритмы? ..... 485**
13.1. Список вопросов для разработчика алгоритмов ..... 486
13.2. Подготовка к собеседованию в технологических компаниях ..... 489

**ЧАСТЬ II. КАТАЛОГ АЛГОРИТМИЧЕСКИХ ЗАДАЧ ..... 493**

**Глава 14. Описание каталога ..... 495**
14.1. Предостережения ..... 496

**Глава 15. Структуры данных ..... 497**
15.1. Словари ..... 497
15.2. Очереди с приоритетами ..... 503
15.3. Суффиксные деревья и массивы ..... 507
15.4. Графы ..... 511
15.5. Множества ..... 515
15.6. Kd-деревья ..... 519

**Глава 16. Численные задачи ..... 525**
16.1. Решение системы линейных уравнений ..... 526
16.2. Уменьшение ширины ленты матрицы ..... 530
16.3. Умножение матриц ..... 532
16.4. Определители и перманенты ..... 535
16.5. Условная и безусловная оптимизация ..... 538
16.6. Линейное программирование ..... 542
16.7. Генерирование случайных чисел ..... 547
16.8. Разложение на множители и проверка чисел на простоту ..... 552
16.9. Арифметика произвольной точности ..... 555
16.10. Задача о рюкзаке ..... 560
16.11. Дискретное преобразование Фурье ..... 565

**Глава 17. Комбинаторные задачи ..... 569**
17.1. Сортировка ..... 570
17.2. Поиск ..... 575
17.3. Поиск медианы и выбор элементов ..... 579
17.4. Генерирование перестановок ..... 582
17.5. Генерирование подмножеств ..... 587
17.6. Генерирование разбиений ..... 590
17.7. Генерирование графов ..... 595
17.8. Календарные вычисления ..... 599
17.9. Календарное планирование ..... 601
17.10. Выполнимость ..... 605

**Глава 18. Задачи на графах с полиномиальным временем исполнения ..... 609**
18.1. Компоненты связности ..... 610
18.2. Топологическая сортировка ..... 613
18.3. Минимальные остовные деревья ..... 617
18.4. Поиск кратчайшего пути ..... 622
18.5. Транзитивное замыкание и транзитивная редукция ..... 627
18.6. Паросочетание ..... 630
18.7. Задача поиска эйлерова цикла и задача китайского почтальона ..... 634
18.8. Реберная и вершинная связность ..... 637
18.9. Потоки в сети ..... 641
18.10. Рисование графов ..... 644
18.11. Рисование деревьев ..... 649
18.12. Планарность ..... 652

**Глава 19. NP-сложные задачи на графах ..... 655**
19.1. Задача о клике ..... 655
19.2. Независимое множество ..... 658
19.3. Вершинное покрытие ..... 661
19.4. Задача коммивояжера ..... 664
19.5. Гамильтонов цикл ..... 668
19.6. Разбиение графов ..... 672
19.7. Вершинная раскраска ..... 675
19.8. Реберная раскраска ..... 679
19.9. Изоморфизм графов ..... 680
19.10. Дерево Штейнера ..... 685
19.11. Разрывающее множество ребер или вершин ..... 689

**Глава 20. Вычислительная геометрия ..... 693**
20.1. Элементарные задачи вычислительной геометрии ..... 693
20.2. Выпуклая оболочка ..... 698
20.3. Триангуляция ..... 702
20.4. Диаграммы Вороного ..... 706
20.5. Поиск ближайшей точки ..... 709
20.6. Поиск в области ..... 713
20.7. Местоположение точки ..... 716
20.8. Выявление пересечений ..... 720
20.9. Разложение по контейнерам ..... 724
20.10. Преобразование к срединной оси ..... 728
20.11. Разбиение многоугольника на части ..... 731
20.12. Упрощение многоугольников ..... 734
20.13. Выявление сходства фигур ..... 737
20.14. Планирование перемещений ..... 740
20.15. Конфигурации прямых ..... 744
20.16. Сумма Минковского ..... 747

**Глава 21. Множества и строки ..... 750**
21.1. Поиск покрытия множества ..... 750
21.2. Задача укладки множества ..... 755
21.3. Сравнение строк ..... 758

21.4. Нечеткое сравнение строк ... 761
21.5. Сжатие текста ... 766
21.6. Криптография ... 770
21.7. Минимизация конечного автомата ... 776
21.8. Максимальная общая подстрока ... 779
21.9. Поиск минимальной общей надстроки ... 782

**Глава 22. Ресурсы ... 786**
22.1. Программные системы ... 786
22.1.1. Библиотека LEDA ... 786
22.1.2. Библиотека CGAL ... 787
22.1.3. Библиотека Boost ... 787
22.1.4. Библиотека Netlib ... 788
22.1.5. Коллекция алгоритмов ассоциации ACM ... 788
22.1.6. Сайты GitHub и SourceForge ... 788
22.1.7. Система Stanford GraphBase ... 789
22.1.8. Пакет Combinatorica ... 789
22.1.9. Программы из книг ... 789
Книга «Programming Challenges» ... 790
Книга «Combinatorial Algorithms for Computers and Calculators» ... 790
Книга «Computational Geometry in C» ... 790
Книга «Algorithms in C++» ... 790
Книга «Discrete Optimization Algorithms in Pascal» ... 791
22.2. Источники данных ... 791
22.3. Библиографические ресурсы ... 791
22.4. Профессиональные консалтинговые услуги ... 792

**Глава 23. Список литературы ... 793**

**Приложение. Описание электронного архива ... 838**

**Предметный указатель ... 839**

# Предисловие

Многие профессиональные программисты не очень хорошо подготовлены к решению проблем разработки алгоритмов. Это прискорбно, так как методика разработки алгоритмов является одной из важнейших *технологий* вычислительной техники.

Эта книга была задумана как руководство по разработке алгоритмов, в котором я планировал изложить технологию разработки комбинаторных алгоритмов. Она рассчитана как на студентов, так и на профессионалов. Книга состоит из двух частей. *Часть I* представляет собой общее введение в *технические* приемы разработки и анализа компьютерных алгоритмов. *Часть II* предназначена для использования в качестве *справочного* и познавательного материала и состоит из каталога алгоритмических ресурсов, реализаций и обширного списка литературы.

## Читателю

Меня очень обрадовал теплый прием, с которым были встречены предыдущие издания книги «Алгоритмы. Руководство по разработке». За время с выхода в свет первого издания книги в 1997 г. было продано свыше 60 тысяч ее экземпляров в различных форматах. Ее перевели на китайский, японский и русский языки. Книга была признана уникальным руководством по применению алгоритмических приемов для решения задач, которые часто возникают в реальной жизни. Но со времени опубликования второго издания в 2008 г. многое изменилось в этом мире. По мере того как компании по разработке программного обеспечения в процессе собеседований при приеме на работу уделяли все большее внимание вопросам по алгоритмам, популярность моей книги взлетела до небес. Многие успешные соискатели использовали ее для подготовки к таким собеседованиям.

Хотя, наверное, область разработки алгоритмов является самой классической областью теории вычислительных систем, она продолжает прогрессировать и изменяться. Рандомизированные алгоритмы и структуры данных приобретают все возрастающую важность, в особенности методы на основе хеширования. Последние достижения понизили алгоритмическую сложность наилучших известных алгоритмов решения таких фундаментальных задач, как нахождение минимальных остовных деревьев, изоморфизма графов и потоков в сетях. Если считать, что начало современной разработки и анализа алгоритмов было положено приблизительно в 1970 г., то получается, что около 20% современной истории алгоритмов приходится на время, прошедшее после второго издания руководства.

Поэтому настало время для нового издания книги, отражающего изменения, произошедшие в алгоритмическом и промышленном мире, а также учитывающего отзывы и

пожелания, которые я получил от сотен ее читателей. Мои основные цели в третьем издании следующие:

- в *первой части книги* («Практическая разработка алгоритмов») расширить такие важные предметы, как хеширование, рандомизированные алгоритмы, алгоритмы типа «разделяй и властвуй», аппроксимирующие алгоритмы, а также добавить предмет квантовых вычислений;
- во *второй части книги* («Каталог алгоритмических задач») обновить справочный материал для всех задач каталога;
- в целом воспользоваться последними достижениями в области цветной печати, чтобы создать более информативные и привлекающие внимание иллюстрации.

Читатели одобрили три аспекта руководства: каталог алгоритмических задач, истории из жизни и электронную версию книги. Эти элементы сохранены и в настоящем издании.

- *Каталог алгоритмических задач.* Не так-то просто узнать, что уже известно о стоящей перед вами задаче. Именно поэтому в книге имеется каталог 75 наиболее важных задач, часто возникающих в реальной жизни. Просматривая его, студент или специалист-практик может быстро выяснить название своей задачи и понять, что о ней известно и как приступить к ее решению.

  Учитывая последние разработки и результаты исследований, я обновил каждый раздел каталога. Особое внимание было уделено обновлению программных реализаций решений задач с учетом таких источников, как GitHub, которые появились после выхода предыдущего издания.

- *Истории из жизни.* Чтобы продемонстрировать, как алгоритмические задачи возникают в реальной жизни, в материал книги включены неприукрашенные истории, описывающие мой опыт по решению практических задач. При этом преследовалась цель показать, что разработка и анализ алгоритмов представляют собой не отвлеченную теорию, а важный инструмент, требующий умелого обращения.

  В этом издании обновлены самые лучшие жизненные истории из предыдущих изданий книги, а также добавлены новые истории, имеющие отношение к рандомизированным алгоритмам, методу «разделяй и властвуй» и динамическому программированию.

- *Онлайновый компонент.* На моем веб-сайте (**www.algorist.com**) в полном объеме представлены конспекты лекций, а также Wiki-энциклопедия решений задач. Этот веб-сайт был обновлен совместно с книгой. Мои видеолекции по алгоритмам на YouTube получили свыше 900 тысяч просмотров.

Важно обозначить то, чего нет в этой книге. Я не уделяю большого внимания математическому обоснованию алгоритмов и в большинстве случаев ограничиваюсь неформальными рассуждениями. Не найдете вы в этой книге и ни одной теоремы. Более подробную информацию вы можете получить, изучив приведенные программы или справочный материал. Цель этого руководства в том, чтобы как можно быстрее указать читателю верное направление движения.

# Преподавателю

Эта книга содержит достаточно материала для курса «Введение в алгоритмы». Предполагается, что читатель обладает знаниями, полученными при изучении таких курсов, как «Структуры данных» или «Теория вычислительных машин и систем».

На сайте **www.algorist.com** можно загрузить полный набор лекционных слайдов для преподавания материала этой книги. Кроме того, я выложил аудио- и видеолекции с использованием этих слайдов для преподавания полного курса продолжительностью в один семестр. Таким образом, вы можете через Интернет воспользоваться моей помощью в преподавании вашего курса.

С целью облегчить использование книги в качестве учебника она была полностью переработана с учетом следующих моментов:

- *Новый материал.*

  Чтобы отразить последние достижения в области разработки алгоритмов, я добавил новые главы по рандомизированным алгоритмам, алгоритмам типа «разделяй и властвуй» и аппроксимирующим алгоритмам. Я также более углубленно рассмотрел такие темы, как хеширование. Но я также прислушался к читателям, которые умоляли меня держать объем книги в разумных пределах. Поэтому я (с болью в сердце) удалил из нее менее важный материал, чтобы общее количество страниц не более чем на 10% превышало объем предыдущего издания.

- *Более ясное изложение.*

  Перечитывая свой текст десять лет спустя, я испытывал большое удовольствие от того, что многие разделы книги читались легко и непринужденно, но в то же самое время встречались и места, продраться сквозь которые было сродни поиску выхода из лабиринта. Поэтому каждая страница этого издания отредактирована или переписана заново, чтобы обеспечить большую ясность, правильность и логическую связность.

- *Больше ресурсов для собеседований.*

  Книга продолжает пользоваться популярностью для подготовки к собеседованиям при приеме на работу, но мы живем сегодня в быстро меняющемся мире. Поэтому я включил в нее дополнительные новые задачи для собеседований, а также задачи по программированию, связанные с интернет-сервисами по набору персонала LeetCode и Hackerrank. Также добавлен новый раздел с советами, как лучше всего подготовиться к собеседованию при приеме на работу.

- *Остановки для размышлений.*

  Каждую лекцию своего курса я начинаю с «Задачи дня», в которой иллюстрирую свой процесс мышления при решении домашнего задания по теме лекции со всеми неудачными начальными попытками решения и всяким таким прочим. Это издание содержит большее количество разделов «Остановка для размышлений», играющих подобную роль для читателей.

- *Переработанные и новые домашние задания.*

  Это издание книги «Алгоритмы. Руководство по разработке» содержит больше и лучшего качества упражнения для домашней работы, чем предыдущее. Я добавил

свыше сотни новых интересных задач, удалил некоторые менее интересные, а также уточнил упражнения, которые сбивали с толку или были не совсем понятны.

- *Новый формат кода.*

  Во втором издании использовались реализации алгоритмов на языке C, которые заменяли или дополняли описания на псевдокоде. Читатели в целом восприняли их положительно, хотя кое-кто и подверг критике определенные аспекты моего программирования как устаревшие. Так что все программы для этого издания были исправлены и обновлены с выделением цветом синтаксических составляющих.

- *Цветные рисунки.*

  Моя другая книга «The Data Science Design Manual»[1] была издана с цветными рисунками, и меня приятно удивило, насколько это улучшило понимание излагаемых понятий. Теперь и в книге «Алгоритмы. Руководство по разработке» все рисунки цветные. Кроме этого, их содержимое было улучшено в результате экспертного просмотра и оценки.

> ***Примечание от русской редакции***
>
> Стремясь сделать эту книгу не слишком дорогой для российского читателя, будь она выпущена полноцветной, мы избрали для ее издания компромиссный вариант: все рисунки книги, а также листинги программ, содержащие выделения цветом, собраны в отдельные файлы электронного архива, скачать который можно с FTP-сервера издательства «БХВ» по ссылке **https://zip.bhv.ru/9785977567992.zip**, а также со страницы книги на сайте **https://bhv.ru/**. Кроме того, цветные рисунки, содержащие цветовую информацию, важную для понимания материала книги, вынесены на цветную вклейку. Ссылки на такие рисунки помечены в тексте книги префиксом «ЦВ» — например: **Рис. ЦВ-1.1** (см. *приложение*).

# Благодарности

Обновление посвящения каждые десять лет заставляет задуматься о скоротечности времени. За время существования этой книги Рени стала моей женой, а потом матерью наших двух детей, Бонни и Эбби, которые уже больше не дети. Мой отец отошел в мир иной, но моя мама и мои братья Лен и Роб продолжают играть важную роль в моей жизни. Я посвящаю эту книгу членам моей семьи, новым и старым, тем, кто с нами, и тем, кто покинул нас.

Я бы хотел поблагодарить нескольких людей за их непосредственный вклад в новое издание. Майкл Алвин (Michael Alvin), Омар Амин (Omar Amin), Эмили Баркер (Emily Barker) и Джек Женг (Jack Zheng) сыграли неоценимую роль в создании инфраструктуры сайта и при решении разных вопросов по подготовке рукописи. Для предыдущих изданий этими вопросами занимались Рикки Брэдли (Ricky Bradley), Эндрю Гон (Andrew Gaun), Жонг Ли (Zhong Li), Бетсон Томас (Betson Thomas) и Дарио Влах (Dario Vlah). Самый внимательный в мире читатель Роберт Пиче (Robert Pich´e) из Университета города Тампере и студенты Питер Даффи (Peter Duffy), Олеся Елфимова

[1] В русском переводе: «Наука о данных. Учебный курс» (изд-во «Вильямс», 2020). — *Прим. ред.*

(Olesia Elfimova) и Роберт Матцибеккер (Robert Matsibekker) из Университета Стоуни Брук вычитали ранние версии рукописи этого издания и убрали из него множество ошибок, избавив и вас, и меня от этой напасти. Я также выражаю благодарность редакторам издательства Springer-Verlag Уэйну Уиллеру (Wayne Wheeler) и Симону Ризу (Simon Rees).

Многие упражнения были созданы по подсказке коллег или под влиянием других работ. Восстановление первоначальных источников годы спустя является задачей не из легких, но на веб-сайте книги дается ссылка на первоисточник каждой задачи (насколько мне удавалось его вспомнить).

Большую часть моих знаний об алгоритмах я получил, изучая их совместно с моими аспирантами. Многие из них: Йо-Линг Лин (Yaw-Ling Lin), Сундарам Гопалакришнан (Sundaram Gopalakrishnan), Тинг Чен (Ting Chen), Фрэнсин Иванс (Francine Evans), Харальд Рау (Harald Rau), Рики Брэдли (Ricky Bradley) и Димитрис Маргаритис (Dimitris Margaritis) — являются персонажами историй, приведенных в книге. Мне всегда было приятно работать и общаться с моими друзьями и коллегами из Университета Стоуни Брук: Эсти Аркином (Estie Arkin), Майклом Бэндером (Michael Bender), Джингом Ченом (Jing Chen), Резаулом Чоудхури (Rezaul Chowdhury), Джи Гао (Jie Gao), Джо Митчеллом (Joe Mitchell) и Робом Патро (Rob Patro).

## Ограничение ответственности

Традиционно вину за любые недостатки в книге великодушно принимает на себя ее автор. Я же делать этого не намерен. Любые ошибки, недостатки или проблемы в этой книге являются виной кого-то другого, но я буду признателен, если вы поставите меня в известность о них, с тем, чтобы я знал, кто виноват.

*Стивен С. Скиена*

Кафедра вычислительной техники
Университет Стоуни Брук
Стоуни Брук, Нью-Йорк 11794-2424
**http://www.cs.sunysb.edu/~skiena**
Август 2020 г.

# ЧАСТЬ I

# Практическая разработка алгоритмов

# ГЛАВА 1

# Введение в разработку алгоритмов

Что такое алгоритм? Это процедура выполнения определенной задачи. Алгоритм является основополагающей идеей любой компьютерной программы.

Чтобы представлять интерес, алгоритм должен решать общую, корректно поставленную *задачу*. Определение задачи, решаемой с помощью алгоритма, дается описанием всего множества *экземпляров*, которые должен обработать алгоритм, и выхода, т. е. результата, получаемого после обработки одного из этих экземпляров. Описание одного из экземпляров задачи может заметно отличаться от формулировки общей задачи. Например, постановка *задача сортировки* делается таким образом:

**Задача**. Сортировка.

**Вход**. Последовательность из $n$ элементов: $a_1, ..., a_n$.

**Выход**. Перестановка элементов входной последовательности таким образом, что для ее членов справедливо соотношение: $a'_1 \leq a'_2 \leq ... \leq a'_{n-1} \leq a'_n$.

*Экземпляром* задачи сортировки может быть массив имен — например, {Mike, Bob, Sally, Jill, Jan} или набор чисел — например, {154, 245, 568, 324, 654, 324}. Первый шаг к решению — определить, что мы имеем дело с общей задачей, а не с частным случаем.

*Алгоритм* — это процедура, которая принимает любой из возможных входных экземпляров и преобразует его в соответствии с требованиями, указанными в условии задачи. Задачу сортировки можно решить посредством многих разных алгоритмов. В качестве примера одного из таких алгоритмов можно привести метод *сортировки вставками*. Сортировка этим методом заключается во вставке в требуемом порядке элементов из неотсортированной части списка в отсортированную часть, первоначально содержащую один элемент (являющуюся, таким образом, тривиально созданным отсортированным списком). На рис. 1.1 показано применение этого алгоритма для сортировки определенного экземпляра — строки INSERTIONSORT.

А в листинге 1.1 представлена его реализация на языке C.

**Листинг 1.1. Реализация алгоритма сортировки вставками**

```
void insertion_sort(item_type s[], int n)
{
   int i,j;                    /* Счетчики */
      for (i=1; i<n; i++) {
         j=i;
         while ((j>0) && (s[j] < s[j-1])) {
            swap(&s[j],&s[j-1]);
```

```
            j = j-i;
        }
    }
}
```

Обратите внимание на универсальность этого алгоритма. Его можно применять как для сортировки слов, так и для сортировки чисел, а также и любых других значений, используя соответствующую операцию сравнения (<) для определения, какое из двух значений поставить первым. Можно с легкостью убедиться, что этот алгоритм правильно сортирует любой возможный набор входных величин в соответствии с нашим определением задачи сортировки.

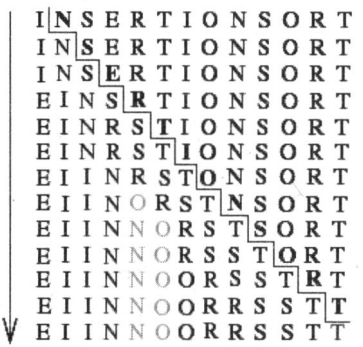



**Рис. 1.1.** Пример использования алгоритма сортировки вставками (шкала времени направлена вниз)

Хороший алгоритм должен обладать тремя свойствами: быть *корректным*, *эффективным* и *легкореализуемым*. Получение комбинации всех трех свойств сразу может оказаться недостижимой задачей. В производственной обстановке любая программа, которая предоставляет достаточно хорошие результаты и не замедляет работу системы, в большинстве случаев является приемлемой, независимо от того, возможны ли улучшения этих показателей. Так что вопрос получения самых лучших возможных результатов или достижения максимальной эффективности обычно возникает только в случае серьезных проблем с производительностью или с законом.

В этой главе основное внимание уделяется вопросу корректности алгоритмов, а их эффективность рассматривается в *главе 2*. Способность определенного алгоритма правильно решить поставленную задачу, т. е. его корректность, редко поддается очевидной оценке. Алгоритмы обычно сопровождаются доказательством их правильности в виде объяснения, *почему* для каждого экземпляра задачи будет выдан требуемый результат. Но прежде чем продолжить обсуждение темы, важно продемонстрировать, что аргумент «это очевидно» никогда не является достаточным доказательством правильности алгоритма.

# 1.1. Оптимизация маршрута робота

Рассмотрим задачу, которая часто возникает на производстве и транспорте. Допустим, что нам нужно запрограммировать роботизированный манипулятор, выполняющий

операцию припаивания контактов интегральной схемы к контактным площадкам на печатной плате. Чтобы запрограммировать манипулятор для выполнения этой задачи, сначала нужно установить порядок, в котором манипулятор припаивает первый контакт, потом второй, третий и т. д., пока не будут припаяны все. После обработки последнего контакта манипулятор возвращается к исходной позиции первого контакта для обработки следующей платы. Таким образом, проход манипулятора представляет собой замкнутый маршрут, или цикл.

Поскольку роботы являются дорогостоящими устройствами, мы хотим минимизировать время, затрачиваемое манипулятором на обработку платы. Будет логичным предположить, что манипулятор перемещается с постоянной скоростью, — соответственно время перемещения от одной точки к другой прямо пропорционально расстоянию между точками. То есть нам нужно найти алгоритмическое решение такой задачи:

**Задача**. Оптимизация маршрута робота.

**Вход**. Множество $S$ из $n$ точек, лежащих на плоскости.

**Выход**. Самый короткий маршрут посещения всех точек множества $S$.

Прежде чем приступать к программированию маршрута манипулятора, нам нужно разработать алгоритм решения этой задачи. На ум может прийти несколько подходящих алгоритмов. Но самым подходящим будет *эвристический алгоритм ближайшего соседа* (nearest-neighbor heuristic). Начиная с какой-либо точки $p_0$, мы идем к ближайшей к ней точке (соседу) $p_1$. От точки $p_1$ мы идем к ее ближайшему еще не посещенному соседу, таким образом исключая точку $p_0$ из числа кандидатов на посещение. Процесс повторяется до тех пор, пока не останется ни одной не посещенной точки, после чего мы возвращаемся в точку $p_0$, завершая маршрут. Псевдокод эвристического алгоритма ближайшего соседа представлен в листинге 1.2.

**Листинг 1.2. Эвристический алгоритм ближайшего соседа**

```
NearestNeighbor(P)
  Из множества P выбираем и посещаем произвольную начальную точку p0
  p = p0
  i = 0
  Пока остаются непосещенные точки
    i = i + 1
    Выбираем и посещаем непосещенную точку pi, ближайшую к точке pi-1
    Посещаем точку pi
  Возвращаемся в точку p0 от точки pn-1
```

Этот алгоритм можно рекомендовать к применению по многим причинам. Он прост в понимании и легко реализуется. Вполне логично сначала посетить близлежащие точки, чтобы сократить общее время прохождения маршрута. Алгоритм дает отличные результаты для входного экземпляра, показанного на рис. ЦВ-1.2[1].

[1] Напомним, что рисунки, помеченные префиксом «ЦВ», вынесены на цветную вклейку, и смотреть их следует там. Кроме того, все рисунки книги, а также листинги программ, содержащие выделения цветом, собраны в отдельные файлы электронного архива, скачать который можно с FTP-сервера издательства «БХВ» по ссылке **https://zip.bhv.ru/9785977567992.zip**, а также со страницы книги на сайте **https://bhv.ru/**. — *Прим. ред.*

Алгоритм ближайшего соседа достаточно эффективен, т. к. в нем каждая пара точек ($p_i$, $p_j$) рассматривается, самое большее, два раза: первый раз при добавлении в маршрут точки $p_i$, а второй — при добавлении точки $p_j$. При всех этих достоинствах алгоритм имеет всего лишь один недостаток — он совершенно неправильный.

*Неправильный?* Да как он может быть неправильным? Поясню: несмотря на то что алгоритм всегда создает маршрут, этот маршрут не обязательно будет самым коротким возможным маршрутом или хотя бы приближающимся к таковому. Рассмотрим множество точек, расположенных в линию, как показано на рис. ЦВ-1.3.

Цифры на рисунке обозначают расстояние от начальной точки до соответствующей точки справа или слева. Начав обход с точки 0 и посещая затем ближайшего непосещенного соседа текущей точки, мы будем метаться вправо-влево через нулевую точку (рис. ЦВ-1.3, *а*), поскольку алгоритм не содержит указания, что нужно делать в случае одинакового расстояния между точками (представьте себе реакцию вашего начальника при виде манипулятора, мечущегося вправо-влево при выполнении такой простой задачи). Гораздо лучшее (более того, оптимальное) решение этого экземпляра задачи — начать обход с крайней левой точки и двигаться направо, посещая каждую точку, после чего возвратиться в исходное положение (рис. ЦВ-1.3, *б*).

Можно сказать, что здесь проблема заключается в неудачном выборе отправной точки маршрута. Почему бы, действительно, не начать маршрут с самой левой точки в качестве точки $p_0$? Это даст нам оптимальное решение этого экземпляра задачи.

Верно на все 100%, но лишь до тех пор, пока мы не развернем множество точек на 90 градусов, сделав все точки самыми левыми. А если к тому же немного сдвинуть первоначальную точку 0 влево, то она опять будет выбрана в качестве отправной. Теперь вместо дергания из стороны в сторону манипулятор станет скакать вверх-вниз, но время прохождения маршрута по-прежнему оставляет желать лучшего. Таким образом, независимо от того, какую точку мы выберем в качестве исходной, алгоритм ближайшего соседа обречен на неудачу с некоторыми экземплярами задачи (т. е. с некоторыми множествами точек), и нам нужно искать другой подход.

Условие, заставляющее всегда искать ближайшую точку, является излишне ограничивающим, поскольку оно принуждает нас выполнять нежелательные переходы. Задачу можно попробовать решить иным способом — соединяя пары самых близких точек, если такое соединение не вызывает никаких проблем, — например, досрочного завершения цикла. Каждая вершина при этом рассматривается как самостоятельная одновершинная цепочка. Соединив все их вместе, мы получим одну цепочку, содержащую все точки. Соединив две конечные точки, мы получим цикл. На любом этапе выполнения этого *эвристического алгоритма ближайших пар* у нас имеется множество отдельных вершин и концов цепочек, не имеющих общих вершин, которые можно соединить. Псевдокод соответствующего алгоритма приведен в листинге 1.3.

**Листинг 1.3. Эвристический алгоритм ближайших пар**

```
ClosestPair(P)
  Пусть n — количество точек множества P.
  For i = 1 to n — 1 do
    d = ∞
```

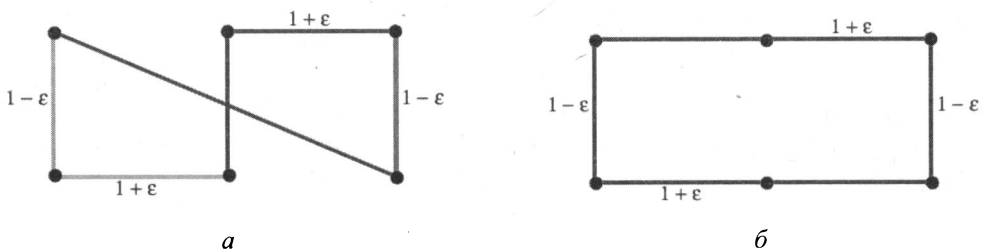
```
    Для каждой пары точек(s, t), не имеющих общих вершин цепей
      if dist(s, t) ≤ d then s_m = s, t_m = t и d = dist(s, t)
  Соединяем две конечные точки (s_m, t_m) ребром
```

Для экземпляра задачи, представленного на рис. ЦВ-1.3, этот алгоритм работает должным образом. Сначала точка 0 соединяется со своими ближайшими соседями, точками 1 и –1. Потом соединение следующих ближайших пар точек выполняется поочередно влево-вправо, расширяя центральную часть по одному сегменту за проход. Эвристический алгоритм ближайших пар чуть более сложный и менее эффективный, чем предыдущий, но, по крайней мере, для нашего экземпляра задачи он дает правильный результат.

Впрочем, это верно не для всех экземпляров. Посмотрите на результаты работы алгоритма на рис. 1.4, *а*. Этот входной экземпляр состоит из двух рядов равномерно расположенных точек. Расстояние между рядами $(1 - \varepsilon)$ несколько меньше, чем расстояние между смежными точками в рядах $(1 + \varepsilon)$. То есть пары наиболее близких точек располагаются не по периметру, а напротив друг друга. Сначала противоположные точки соединяются попарно, а затем полученные пары соединяются поочередно по периметру. Общее расстояние маршрута алгоритма ближайших пар в этом случае будет равно

$$3(1-\varepsilon)+2(1+\varepsilon)+\sqrt{(1-\varepsilon)^2+(2+2\varepsilon)^2}$$

Этот маршрут на 20% длиннее, чем маршрут на рис. 1.4, *б*, когда $\varepsilon \to 0$. Более того, есть входные экземпляры, дающие значительно худшие результаты, чем этот.




*а* *б*

**Рис. 1.4.** Неудачный входной экземпляр для алгоритма ближайших пар *(а)* и оптимальное решение *(б)*

Таким образом, этот алгоритм тоже не годится. Какой из этих двух алгоритмов более эффективный? На этот вопрос нельзя ответить, просто посмотрев на них. Но очевидно, что оба алгоритма могут выдать очень плохие маршруты на некоторых с виду простых входных экземплярах.

Но каков же правильный алгоритм решения этой задачи? Можно попробовать перечислить *все* возможные перестановки множества точек, а потом выбрать перестановку, сводящую к минимуму длину маршрута. Псевдокод этого алгоритма приведен в листинге 1.4.

**Листинг 1.4. Оптимальный алгоритм поиска маршрута**

```
OptimalTSP(P)
  d = ∞
```

```
Для каждой перестановки Pi из общего числа перестановок n! множества точек P
  If (cost(Pi) ≤ d) then d = cost(Pi) и Pmin = Pi
Return Pmin
```

Так как рассматриваются все возможные упорядочения, то получение самого короткого маршрута гарантировано. Поскольку мы выбираем самую лучшую комбинацию, алгоритм правильный. В то же самое время он чрезвычайно медленный, и даже очень мощный компьютер не сможет в течение дня рассчитать все

$$20! = 2\,432\,902\,008\,176\,640\,000$$

возможных перестановок 20 точек. А о реальных ситуациях, когда количество точек печатной платы достигает тысячи, можно и не говорить. Все компьютеры в мире, работая круглосуточно, не смогут даже приблизиться к решению этой задачи до конца существования Вселенной, а тогда решение этой задачи, скорее всего, уже не будет актуальным.

Поиском эффективного алгоритма решения этой задачи, называющейся *задачей коммивояжера* (Traveling Salesman Problem, TSP), мы будем заниматься на протяжении большей части этой книги. Если же вам не терпится узнать решение уже сейчас, то вы можете посмотреть его в *разд. 19.4*.

#### Подведение итогов

*Алгоритмы*, т. е. процедуры, которые всегда выдают правильное решение, коренным образом отличаются от *эвристик*, которые обычно выдают достаточно хорошие результаты, но не гарантируют их правильность.

## 1.2. Задача календарного планирования

Теперь рассмотрим задачу календарного планирования. Представьте, что вы — кинозвезда, и вам наперебой предлагают роли в разных кинофильмах, общее количество которых равно *n*. Каждое предложение имеет условие, что вы должны посвятить себя ему с первого до последнего дня съемок. Поэтому вы не можете сниматься одновременно в фильмах с полностью или частично накладывающимися периодами съемок.

Ваш критерий для принятия того или иного предложения довольно прост: вы хотите заработать как можно больше денег. Поскольку вам платят одинаково за каждый фильм, то вы стремитесь получить роли в как можно большем количестве фильмов, периоды съемок которых не конфликтуют. На рис. 1.5 показаны фильмы, в которых вам предлагают роли.

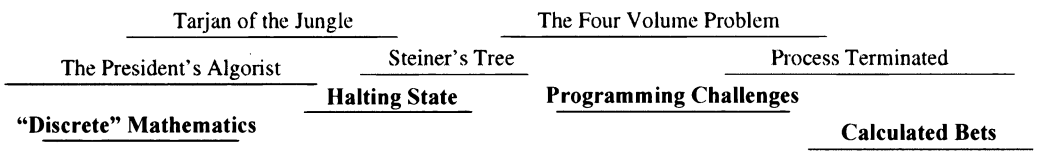



**Рис. 1.5.** Экземпляр задачи планирования непересекающихся календарных периодов. Оптимальное решение представлено четырьмя названиями, выделенными полужирным шрифтом

Как можно видеть, вы можете сниматься, самое большее, в четырех фильмах: «"Discrete" Mathematics», «Programming Challenges» и «Calculated Bets», а также или в «Halting State», или в «Steiner's Tree». А в менее очевидных случаях вам (или вашему менеджеру) нужно будет решить следующую алгоритмическую задачу календарного планирования:

**Задача**. Планирование съемок в фильмах.

**Вход.** Множество $I$ интервалов времени $n$ в линейном порядке.

**Выход.** Самое большое подмножество непересекающихся интервалов времени, которое возможно в множестве $I$.

На ум может прийти несколько способов решения этой задачи. Один из них основан на представлении, что надо работать всегда, когда это возможно. Это означает, что вам нужно брать роль в фильме, съемки которого начинаются раньше всех других. Псевдокод этого алгоритма представлен в листинге 1.5.

**Листинг 1.5. Алгоритм первой возможной работы**

```
EarliestJobFirst(I)
    Из множества фильмов I берем роль в фильме j с самым ранним началом
    съемок, период которых не пересекается с периодом ваших предыдущих
    обязательств. Поступаем таким образом до тех пор, пока больше
    не останется таких фильмов.
```

Такой подход выглядит логично, по крайней мере, до тех пор, пока вы не осознаете, что, хватаясь за самую раннюю работу, можете пропустить несколько других, если первый фильм является сериалом. Пример такой ситуации показан на рис. 1.6, *а*, где самым ранним фильмом является киноэпопея «War and Peace», которая закрывает перед вами все другие перспективы.




*а* *б*

**Рис. 1.6.** Неудачные экземпляры задач для применения эвристики:
*а* — самая ранняя работа первая; *б* — самая короткая работа первая.
Оптимальное решение представлено более толстыми линиями

Этот пример заставляет искать другое решение. Проблема с фильмом «War and Peace» заключается в том, что его съемки длятся слишком долго. В таком случае, может быть, наоборот, вам следует брать роли только в фильмах с самыми короткими периодами съемок? Разве не очевидно, что чем быстрее вы закончите сниматься в одном фильме, тем раньше сможете начать сниматься в другом, максимизируя таким образом количество фильмов в любой выбранный период времени? Псевдокод этого эвристического алгоритма представлен в листинге 1.6.

**Листинг 1.6. Алгоритм самого короткого периода**

```
ShortestJobFirst(I)
  While (I ≠ ∅) do
    Из всего множества фильмов I берем фильм j
    с самым коротким периодом съемок
    Удаляем фильм j и любой другой фильм, период съемок которого
    пересекается с фильмом j, из множества доступных фильмов I
```

Но и этот подход окажется действенным только до тех пор, пока вы не увидите, что можете упустить возможность заработать больше (рис. 1.6, *б*). Хотя в этом случае потери меньше, чем в предыдущем случае, тем не менее вы получите только половину возможных заработков.

На этом этапе может показаться заслуживающим внимания алгоритм, который перебирает все возможные комбинации. Как и в случае с задачей путешествующего коммивояжера, исчерпывающий поиск гарантирует правильный результат. Если отвлечься от подробностей проверки подмножеств интервалов (т. е. периодов съемок) на пересечение, этот алгоритм можно выразить псевдокодом, представленным в листинге 1.7.

**Листинг 1.7. Алгоритм полного перебора**

```
ExhaustiveScheduling(I)
  j = 0
  Smax = ∅
  Для каждого из 2^n подмножеств Si множества интервалов I
    If (Si непересекающееся) и (size(Si) > j)
       then j = size(Si) и Smax = Si
  Return Smax
```

Но насколько эффективным будет такой алгоритм? Здесь ключевым ограничением является необходимость выполнить перечисление $2^n$ подмножеств $n$ элементов. А положительным обстоятельством является то, что это *намного* лучше, чем перечисление всех $n!$ порядков $n$ элементов, как предлагается в задаче оптимизации маршрута роботизированного манипулятора. Здесь же при $n = 20$ имеется около миллиона подмножеств, которые можно за несколько секунд перебрать на современном компьютере. Но когда выбор фильмов возрастет до $n = 100$, то количество возможных подмножеств будет равно $2^{100}$, что намного больше, чем значение 20!, которое положило нашего робота на лопатки в предыдущей задаче.

Разница между задачей составления маршрута и задачей календарного планирования заключается в том, что для последней *имеется* алгоритм, который решает задачу и правильно, и эффективно. Для этого вам нужно брать роли только в фильмах с самым ранним окончанием съемок, т. е. выбрать такой временной интервал $x$, у которого правая конечная точка находится левее правых конечных точек всех прочих временных интервалов. Таким фильмом в нашем примере (см. рис. 1.5) является «“Discrete” Mathematics». Вполне возможно, что съемки других фильмов начались раньше, чем съемки фильма $x$, но все они должны пересекаться друг с другом (по крайней мере, частично), поэтому мы можем выбрать, самое большее, один фильм из всей группы.

Съемки фильма $x$ закончатся раньше всего, поэтому остальные фильмы с накладывающимися съемочными периодами потенциально блокируют другие возможности, расположенные справа от них. Очевидно, что, выбрав фильм $x$, вы никак не можете проиграть. Псевдокод эффективного алгоритма правильного решения задачи календарного планирования будет выглядеть так, как показано в листинге 1.8.

**Листинг 1.8. Оптимальный алгоритм календарного планирования**

```
OptimalScheduling(I)
  While (I ≠ ∅) do
    Из всего множества фильмов I выбираем фильм j с самым
        ранним окончанием съемок
    Удаляем фильм j и любой другой фильм, съемки которого
        пересекаются с фильмом j, из множества доступных фильмов I
```

Обеспечение оптимального решения для всего диапазона возможных входных экземпляров представляется трудной, но, как правило, выполнимой задачей. Важной частью процесса разработки такого алгоритма является поиск входных экземпляров задачи, которые опровергают наше допущение о правильности алгоритма-претендента на решение. Эффективные алгоритмы часто скрываются где-то совсем рядом, и эта книга поможет вам развить навыки их обнаружения.

#### Подведение итогов

Кажущиеся вполне логичными алгоритмы очень легко могут оказаться неправильными. Правильность алгоритма требует тщательного доказательства.

## 1.3. Обоснование правильности алгоритмов

Будем надеяться, что предшествующие примеры продемонстрировали вам всю сложность темы правильности алгоритмов. Правильные алгоритмы выделяются из общего числа с помощью специальных инструментов, главный из которых называется *доказательством*.

Адекватное математическое доказательство состоит из нескольких частей. Прежде всего, требуется ясная и четкая формулировка того, что вы пытаетесь доказать. Потом необходим набор предположений, которые всегда считаются верными и поэтому могут использоваться как часть доказательства. Далее следует цепь умозаключений, которая приводит нас от начальных предположений к конечному утверждению, которое мы пытаемся доказать. Наконец, небольшой черный квадрат ■ в тексте указывает на конец доказательства.

В этой книге формальным доказательствам не уделяется большого внимания, поскольку правильное формальное доказательство привести очень трудно, а неправильное может вас сильно дезориентировать. На самом деле, доказательство сводится *демонстрации*. Доказательства полезны только тогда, когда они простые и незамысловатые, и представляют собой ясные и лаконичные аргументы, объясняющие, почему алгоритм удовлетворяет требованию нетривиальной правильности.

Правильные алгоритмы требуют тщательного изложения и определенных усилий для доказательства как их правильности, так и того факта, что они *не* являются неправильными.

### 1.3.1. Задачи и свойства

Прежде чем приступать к разработке алгоритма, необходимо тщательно сформулировать задачу, которая подлежит решению.

Постановка задачи состоит из двух частей: набора допустимых входных экземпляров и требований к выходу алгоритма. Невозможно доказать правильность алгоритма для нечетко поставленной задачи. Иными словами, поставьте неправильно задачу, и вы получите неправильный ответ.

Постановки некоторых задач допускают слишком широкий диапазон входных экземпляров. Допустим, что в нашей задаче календарного планирования съемки фильмов могут прерываться на некоторое время. Например, съемки фильма могут быть запланированы на сентябрь и ноябрь, а октябрь свободен. Тогда календарный план съемок для любого фильма будет состоять из *набора* временны́х интервалов. В этом случае мы можем браться за роли в двух фильмах с чередующимися, но не пересекающимися периодами съемок. Такую общую задачу календарного планирования нельзя решить с помощью алгоритма самого раннего окончания съемок. Более того, для решения этой общей задачи вообще *не существует* эффективного алгоритма, как мы увидим в *разд. 11.3.2*.

> ***Подведение итогов***
>
> Важным и заслуживающим внимания приемом разработки алгоритмов является сужение множества допустимых экземпляров задачи до тех пор, пока не будет найден правильный и эффективный алгоритм. Например, задачу общих графов можно свести к задаче деревьев, а двумерную геометрическую задачу — к одномерной.

При указании требований выхода алгоритма часто допускаются две ошибки. Первая из них — плохая формулировка вопроса. Примером может служить вопрос о *наилучшем* маршруте между двумя точками на карте при отсутствии определения, что значит «наилучший». Лучший в каком смысле? В смысле самого короткого расстояния, самого короткого времени прохождения или, может быть, минимального количества поворотов? Все эти критерии вполне могут требовать разных решений.

Второй ошибкой является формулирование составных целей. Каждый из трех только что упомянутых критериев наилучшего маршрута является четко определенной целью правильного эффективного алгоритма оптимизации. Но в качестве требования к решению из этих критериев можно выбрать только один. Например, формулировка: «Найти самый короткий маршрут от точки к точке, содержащий количество поворотов, не более чем в два раза превышающее необходимое», является вполне четкой постановкой задачи. Но решить такую задачу очень трудно, т. к. решение требует сложного анализа.

Для примеров постановки задач читателю настоятельно рекомендуется ознакомиться с постановкой каждой из 75 задач из *второй части* этой книги. Правильная постановка задачи является важной частью ее решения. Изучение постановок всех этих классиче-

ских задач поможет вам распознать задачи, постановкой и решением которых кто-то уже занимался до вас.

## 1.3.2. Представление алгоритмов

Цепь логических умозаключений об алгоритме невозможно построить без тщательного описания последовательности шагов, которые необходимо выполнить. Для этой цели наиболее часто употребляются, по отдельности или в совокупности, три формы представления алгоритма: обычный язык, псевдокод и язык программирования. Самым загадочным из этих средств представления алгоритма является псевдокод — это средство лучше всего можно определить как язык программирования, который никогда не выдает сообщений о синтаксических ошибках. Все три способа являются полезными, т. к. существует естественное стремление к компромиссу между легкостью восприятия и точностью представления алгоритма. Наиболее простым для понимания языком представления алгоритма является обычный язык, но в то же время он наименее точен. С другой стороны, такие языки, как Java или C/C++, позволяют точно выразить алгоритм, но создавать и понимать алгоритмы на этих языках — задача не из легких. В отношении сложности применения и понимания псевдокод представляет золотую середину между этими двумя крайностями.

Выбор самого лучшего способа представления алгоритма зависит от ваших предпочтений. Я, например, сначала описываю свои алгоритмические идеи на обычном языке (с иллюстрациями!), а затем перехожу на более формальный псевдокод, более близкий к языку программирования, или даже на настоящий язык программирования для уточнения сложных деталей.

Не допускайте ошибку, которую часто делают мои студенты, — используют псевдокод, чтобы приукрасить плохо определенную идею и придать ей более формальный и солидный вид. При описании алгоритма следует стремиться к ясности. Например, алгоритм ExhaustiveScheduling (см. листинг 1.7) можно было бы выразить на обычном языке так, как показано в листинге 1.9.

**Листинг 1.9. Алгоритм полного перебора**

```
ExhaustiveScheduling(I)
  Протестировать все 2^n подмножеств множества I и возвратить самое
    большое подмножество непересекающихся интервалов.
```

#### *Подведение итогов*

В основе любого алгоритма лежит *идея*. Если в описании алгоритма ваша идея ясно не просматривается, значит, вы используете для ее выражения нотацию слишком низкого уровня.

## 1.3.3. Демонстрация неправильности алгоритма

Самый лучший способ доказать *неправильность* алгоритма — найти экземпляр задачи, для которого алгоритм выдает неправильный ответ. Такие экземпляры задачи называются *контрпримерами*. Никто в здравом уме не будет защищать алгоритм, для которо-

го был предоставлен контрпример. Вполне разумно выглядящие алгоритмы можно моментально опровергнуть посредством очень простых контрпримеров. Хороший контрпример должен обладать двумя важными свойствами:

- *проверяемостью.*

  Чтобы продемонстрировать, что некий входной экземпляр задачи является контрпримером для определенного алгоритма, требуется вычислить ответ, который алгоритм выдаст для этого экземпляра, и предоставить лучший ответ с тем, чтобы доказать, что алгоритм не смог его найти;

- *простотой.*

  Хороший контрпример не содержит ничего лишнего и ясно демонстрирует, *почему* именно предлагаемый алгоритм неправильный. Простота контрпримера важна по той причине, что его обоснование нужно выполнять мысленно. Поэтому обнаруженный контрпример следует упростить. Контрпример на рис. 1.6, *а* можно упростить и улучшить, сократив количество пересекающихся периодов с пяти до двух.

Развитие навыков поиска контрпримеров будет стоить затраченного времени. Этот процесс в чем-то подобен разработке наборов тестов для проверки компьютерных программ, но в нем главную роль играет удачная догадка, а не перебор вариантов. Приведу несколько советов.

- *Ищите мелкомасштабные решения.*

  Обратите внимание, что мои контрпримеры для задач поиска маршрута содержат шесть или меньше точек, а для задач календарного планирования — только три временных интервала. Это обстоятельство указывает на то, что если алгоритм неправильный, то доказать это можно на очень простом экземпляре. Алгоритмисты-любители нередко создают большой запутанный экземпляр, с которым потом не могут справиться. А профессионалы внимательно изучат несколько небольших экземпляров, т. к. их легче осмыслить и проверить.

- *Рассмотрите все решения.*

  Для первого наименьшего нетривиального значения $n$ обычно имеется только небольшое количество возможных экземпляров. Например, существуют только три представляющих интерес способа расположения двух интервалов на прямой: неперекрывающиеся интервалы, частично перекрывающиеся интервалы и полностью перекрывающиеся интервалы. Все случаи размещения на прямой трех интервалов (включая контрпримеры для обоих эвристических алгоритмов календарного планирования) можно создать, по-разному добавляя третий интервал к этим двум, расположенным указанными тремя способами.

- *Ищите слабое звено.*

  Если рассматриваемый вами алгоритм работает по принципу «всегда берем самое большее» (так называемый *жадный алгоритм*), то подумайте, почему этот подход может быть неправильным.

- *Ищите ограничения.*

  «Жадный» эвристический алгоритм можно сбить с толку необычным методом предоставления входного экземпляра, содержащего одинаковые элементы. В этой

ситуации алгоритму будет не на чем основывать свое решение, и он, возможно, возвратит неоптимальное решение.

- *Рассматривайте крайние случаи.*

Многие контрпримеры представляют собой комбинацию большого и малого, правого и левого, многих и немногих, далекого и близкого. Обычно легче осмыслить и проверить примеры по краям диапазона, чем из его середины. Рассмотрим в качестве примера два скопления точек, расстояние $d$ между которыми намного превышает расстояние между точками в любом из них. Длина оптимального маршрута коммивояжера в этой ситуации будет практически равна $2d$, независимо от количества посещаемых точек, т. к. длина маршрута внутри каждого скопления точек несущественна.

#### Подведение итогов

Лучший способ опровергнуть правильность эвристического алгоритма — испытать его на контрпримерах.

### Остановка для размышлений: «Жадные» кинозвезды?

**ЗАДАЧА.** Вспомним задачу календарного планирования, в которой нам нужно определить наибольшее возможное множество непересекающихся временных интервалов в заданном множестве $S$. Для решения этой задачи предлагается использовать естественный «жадный» эвристический алгоритм, который выбирает интервал $i$, перекрывающий наименьшее количество других интервалов в множестве $S$, и удаляет эти интервалы. Процесс повторяется до тех пор, пока не останется свободных интервалов. Приведите контрпример для этого предлагаемого алгоритма.

**РЕШЕНИЕ.** Возможный контрпример приводится на рис. 1.7.



**Рис. 1.7.** Контрпример для «жадного» эвристического алгоритма календарного планирования. Выбор пунктирного интервала, который перекрывает наименьшее количество других интервалов, блокирует оптимальное решение (четыре штрихпунктирных интервала)

Наибольшее возможное независимое множество содержит четыре интервала (обозначенных штрихпунктиром: — · — · — ·), но два из этих интервалов перекрываются самым коротким интервалом (обозначенным пунктиром: • • • •). Взяв его, мы обречены на решение, состоящее всего лишь из трех интервалов. Но каким образом подойти к созданию такого примера? Мой процесс мышления в этом направлении начался с создания цепочки с нечетным количеством интервалов, каждый из которых перекрывает один другой интервал слева и один справа. Выбор цепочки с четным количеством интервалом нарушит оптимальное решение (*ищем слабое звено*). Каждый из интерва-

лов перекрывает два других интервала, за исключением самого левого и самого правого интервалов (*ищем ограничения*). Чтобы сделать эти конечные интервалы непривлекательными, поверх них можно навалить кучу других интервалов (*ищем крайности*). Наша цепочка содержит наименьшее количество интервалов (7), при которых эта конструкция работоспособна. ■

## 1.3.4. Индукция и рекурсия

Один факт, что для только что рассмотренного алгоритма не был найден контрпример, вовсе не означает, что алгоритм правильный. Для этого требуется доказательство, или демонстрация правильности. Для доказательства правильности алгоритма часто применяется математическая индукция.

Мои первые впечатления о математической индукции были таковы, словно это какое-то шаманство. Вы берете формулу типа:

$$\sum_{i=1}^{n} i = n(n+1)/2,$$

доказываете ее для базового случая — например, $n = 1$ или 2, потом *допускаете*, что утверждение справедливо для $n - 1$, и на основе этого допущения доказываете, что формула действительно верна для общего $n$. И это называется доказательством? Полнейший абсурд!

Мои первые впечатления о рекурсии в программировании были точно такими же — чистое шаманство. Программа проверяет входной аргумент на равенство базовому значению — например, 1 или 2. При отрицательном результате такой проверки более сложный случай решается путем разбивки его на части и вызова *этой же программы в качестве подпрограммы* для решения этих частей. Это называется программой? Полнейший абсурд!

Причиной, благодаря которой как рекурсия, так и математическая индукция кажутся шаманством, является тот факт, что рекурсия и *есть* математическая индукция. В обеих имеются общие и граничные условия, при этом общее условие разбивает задачу на все более мелкие части, а граничное, или начальное, условие завершает рекурсию. Если вы понимаете одно из двух — или рекурсию, или индукцию, — вы в состоянии понять, как работает другое.

Мне приходилось слышать, что программист — это математик, который умеет строить доказательства только методом индукции. Отчасти дело в том, что программисты — никудышные построители доказательств, но главным образом в том, что многие изучаемые ими алгоритмы являются или рекурсивными, или инкрементными (поэтапными).

Рассмотрим, например, правильность алгоритма сортировки вставками, представленного в начале этой главы. Его правильность можно *обосновать* методом индукции:

- базовый экземпляр содержит всего лишь один элемент, а по определению одноэлементный массив является полностью отсортированным;
- предполагаем, что после первых $n - 1$ итераций сортировки вставками первые $n - 1$ элементов массива $A$ будут полностью отсортированы;

- чтобы определить, куда следует вставить последний элемент $x$, нам нужно найти уникальную ячейку между наибольшим элементом, не превышающим $x$, и наименьшим элементом, большим чем $x$. Для этого мы сдвигаем все бóльшие элементы назад на одну позицию, создавая место для элемента $x$ в требуемой позиции. ■

Но к индуктивным доказательствам нужно относиться с большой осторожностью, т. к. в цепь рассуждений могут вкрасться трудно распознаваемые ошибки. Прежде всего, это *граничные ошибки*. Например, в приведенном ранее доказательстве правильности алгоритма сортировки вставками мы самоуверенно заявили, что между двумя элементами массива $A$ имеется однозначно определяемая ячейка, в которую можно вставить наш элемент $x$, когда массив в нашем базовом экземпляре содержит только одну ячейку. Поэтому для правильной обработки частных случаев вставки минимальных или максимальных элементов необходимо соблюдать бóльшую осторожность.

Другой, более распространенный тип ошибок в индуктивных доказательствах связан с небрежным подходом к расширению экземпляра задачи. Добавление всего лишь одного элемента к экземпляру задачи может полностью изменить оптимальное решение. Соответствующий пример для задачи календарного планирования показан на рис. 1.8.



**Рис. 1.8.** Оптимальное решение (прямоугольники) до и после внесения изменений (обозначены пунктиром) в экземпляр задачи

После добавления нового временнóго интервала в экземпляр задачи новое оптимальное расписание может не содержать ни одного из временных интервалов любого оптимального решения, предшествующего изменению. Бесцеремонное игнорирование таких аспектов может вылиться в очень убедительное доказательство полностью неправильного алгоритма.

#### Подведение итогов

Математическая индукция обычно является верным способом проверки правильности рекурсивного алгоритма.

### Остановка для размышлений: Правильность инкрементных алгоритмов

**ЗАДАЧА**. Доказать правильность рекурсивного алгоритма для инкрементации натуральных чисел, т. е. $y \rightarrow y + 1$, представленного в листинге 1.10.

**Листинг 1.10. Алгоритм для инкрементации натуральных чисел**

```
Increment (y)
  if (y = 0) then return(1) else
    if (y mod 2) = 1 then
      return(2 · Increment(⌊y/2⌋))
    else return(y + 1)
```

**РЕШЕНИЕ**. Лично мне правильность этого алгоритма определенно *не* очевидна. Но т. к. это рекурсивный алгоритм, а я — программист, мое естественное побуждение будет доказать его методом индукции.

Абсолютно очевидно, что алгоритм правильно обрабатывает базовый случай, когда $y = 0$. Совершенно ясно, что $0 + 1 = 1$ и, соответственно, возвращается значение 1.

Теперь допустим, что функция работает правильно для общего случая, когда $y = n - 1$. На основе этого допущения нам нужно продемонстрировать правильность алгоритма для случая, когда $y = n$. Случаи для четных чисел очевидны, поскольку при $(y \bmod 2) = 0$ явно возвращается значение y + 1.

Но для нечетных чисел решение зависит от результата, возвращаемого выражением `Increment(⌊y/2⌋)`. Здесь нам хочется воспользоваться нашим индуктивным допущением, но оно не совсем правильно. Мы сделали допущение, что функция `Increment` работает правильно, когда $y = n - 1$, но для значения $y$, равного приблизительно половине этого значения, мы этого не допускали. Теперь мы можем усилить наше допущение, объявив, что общий случай выдерживается для всех $y \leq n - 1$. Это усиление никоим образом не затрагивает сам принцип, но необходимо, чтобы установить правильность алгоритма.

Теперь случаи нечетных $y$ (т. е. $y = 2m + 1$ для целого числа $m$) можно обработать, как показано в листинге 1.11, решая, таким образом, общий случай.

**Листинг 1.11. Алгоритм для инкрементации нечетных натуральных чисел**

```
2·Increment(⌊(2m + 1)/2⌋) = 2·Increment(⌊m + 1/2⌋)
                          = 2·Increment(m)
                          = 2(m + 1)
                          = 2m + 2 = y + 1
```

■

# 1.4. Моделирование задачи

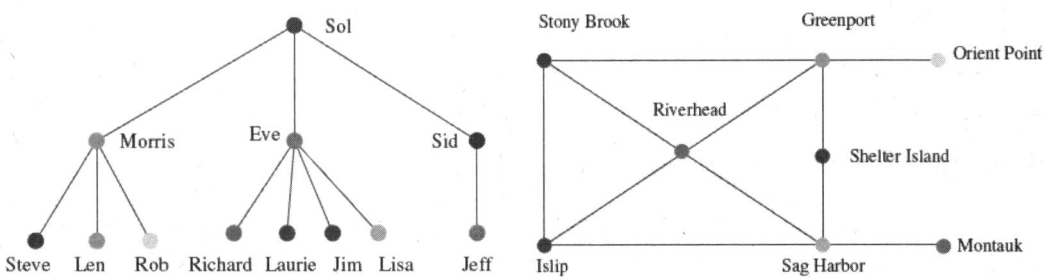
*Моделирование* является искусством формулирования приложения в терминах точно поставленных, хорошо понимаемых задач. Правильное моделирование является ключевым аспектом применения методов разработки алгоритмов решения реальных задач. Более того, правильное моделирование может сделать ненужным разработку или даже реализацию алгоритмов, соотнося ваше приложение с ранее решенной задачей. Правильное моделирование является ключом к эффективному использованию материала *второй части* этой книги.

В реальных приложениях обрабатываются реальные объекты. Вам, может быть, придется работать над системой маршрутизации сетевого трафика, планированием использования классных комнат учебного заведения или поиском закономерностей в корпоративной базе данных. Большинство же алгоритмов спроектированы для работы со строго определенными абстрактными структурами, такими как перестановки, графы или множества. Чтобы извлечь пользу из литературы по алгоритмам, вам нужно научиться выполнять постановку задачи абстрактным образом, в терминах процедур над такими фундаментальными структурами.

## 1.4.1. Комбинаторные объекты

Очень велика вероятность, что над любой интересующей вас алгоритмической задачей уже, скорее всего, работали другие, хотя, возможно, и в совсем иных контекстах. Но не надейтесь узнать, что известно о вашей конкретной «задаче оптимизации процесса *A*», поискав в книге словосочетание «процесс *A*». Вам нужно сначала сформулировать задачу оптимизации вашего процесса в терминах вычислительных свойств стандартных структур, таких как рассматриваемые далее:

- *перестановка* — упорядоченное множество элементов. Например, {1, 4, 3, 2} и {4, 3, 2, 1} являются двумя разными перестановками одного множества целых чисел. Мы уже видели перестановки в задачах оптимизации маршрута манипулятора и календарного планирования. Перестановки будут вероятным исследуемым объектом в задаче поиска «размещения», «маршрута», «границ» или «последовательности»;
- *подмножество* — выборка из множества элементов. Например, множества {1, 3, 4} и {2} являются двумя разными подмножествами множества первых четырех целых чисел. В отличие от перестановок, порядок элементов подмножества не имеет значения, поэтому подмножества {1, 3, 4} и {4, 3, 1} являются одинаковыми. В проблеме календарного планирования нам пришлось иметь дело с подмножествами как вариантами решения задачи. Подмножества будут вероятным исследуемым объектом в задаче поиска «кластера», «коллекции», «комитета», «группы», «пакета» или «выборки»;




*а* *б*

**Рис. 1.9.** Моделирование реальных структур с помощью деревьев (*а*) и графов (*б*)

- *дерево* — иерархическое представление взаимосвязей между объектами. Реальное применение деревьев показано на примере родословного древа семейства Скиена на рис. 1.9, *а*. Деревья будут вероятным исследуемым объектом в задаче поиска «иерархии», «отношений доминирования», «отношений предок/потомок» или «таксономии»;
- *граф* — представление взаимоотношений между произвольными парами объектов. На рис. 1.9, *б* показана модель дорожной сети в виде графа, где вершины представляют населенные пункты, а ребра — дороги, соединяющие населенные пункты. Граф будет вероятным исследуемым объектом в задаче поиска «сети», «схемы», «инфраструктуры» или «взаимоотношений»;

- *точка* — определение места в некотором геометрическом пространстве. Например, расположение автобусных остановок можно описать точками на карте (плоскости). Точка будет вероятным исследуемым объектом в задаче поиска «местонахождения», «позиции», «записи данных» или «расположения»;
- *многоугольник* — определяет область геометрического пространства. Например, с помощью многоугольника (полигона) можно описать границы страны на карте. Многоугольники и многогранники будут вероятными исследуемыми объектами в задаче поиска «формы», «региона», «очертания» или «границы»;
- *строка* — представляет последовательность символов или шаблонов. Например, имена студентов можно представить в виде строк. Строка будет вероятным исследуемым объектом при работе с «текстом», «символами», «шаблонами» или «метками».

Для всех этих фундаментальных структур имеются соответствующие алгоритмы, которые представлены в каталоге задач во *второй части* этой книги. Важно ознакомиться с этими задачами, т. к. они изложены на языке, типичном для моделирования приложений. Чтобы научиться свободно владеть этим языком, просмотрите задачи в каталоге и изучите рисунки входа и выхода для каждой из них. Разобравшись в этих задачах, хотя бы на уровне рисунков и формулировок, вы будете знать, где искать возможный ответ в случае возникновения проблем в разработке вашего приложения.

В книге также представлены примеры успешного применения моделирования приложений в виде описаний решений реальных задач. Однако здесь необходимо высказать одно предостережение. Моделирование сводит разрабатываемое приложение к одной из многих существующих задач и структур. Такой процесс по своей природе является ограничивающим, и некоторые нюансы модели могут не соответствовать вашей конкретной задаче. Кроме того, встречаются задачи, которые можно моделировать несколькими разными способами.

Моделирование является всего лишь первым шагом в разработке алгоритма решения задачи. Внимательно отнеситесь к отличиям вашего приложения от потенциальной модели, но не спешите с заявлением, что ваша задача является уникальной и особенной. Временно игнорируя детали, которые не вписываются в модель, вы сможете найти ответ на вопрос, действительно ли они являются принципиально важными.

#### Подведение итогов

Моделирование разрабатываемого приложения в терминах стандартных структур и алгоритмов является важнейшим шагом в поиске решения.

## 1.4.2. Рекурсивные объекты

Научившись мыслить рекурсивно, вы научитесь определять большие сущности, состоящие из меньших сущностей *точно такого же типа, как и большие*. Например, если рассматривать дом как набор комнат, то при добавлении или удалении комнаты дом остается домом.

В мире алгоритмов рекурсивные структуры встречаются повсеместно. Более того, каждую из ранее описанных абстрактных структур можно считать рекурсивной. На рис. 1.10 видно, как легко они разбиваются на составляющие:

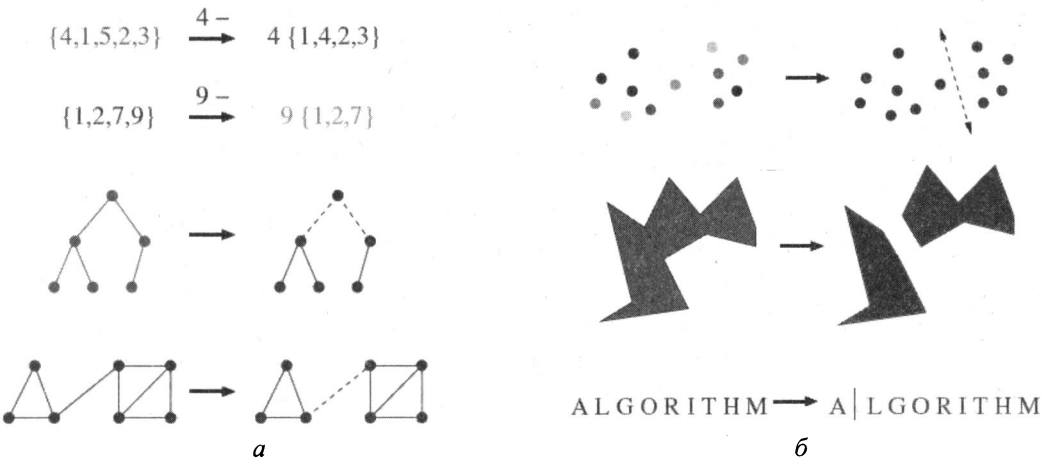



**Рис. 1.10.** Рекурсивное разложение комбинаторных объектов: *а* — перестановки, подмножества, деревья, графы; *б* — множества точек, многоугольники и строки. Обратите внимание на то, что после удаления одного элемента из перестановки {1, ..., *n*} оставшиеся элементы перенумеровываются, чтобы сохранить перестановку {1, ..., *n* – 1} элементов

- *Перестановки.*

  Удалив первый элемент перестановки {1, ..., *n*} *n*-го количества элементов, мы получим перестановку остающихся *n* – 1 элементов. Для этого может потребоваться выполнить перенумерацию, чтобы объект оставался перестановкой последовательных целых чисел. Например, удалив из множества {4, 1, 5, 2, 3} первый элемент и выполнив перенумерацию оставшихся элементов, получим перестановку {1, 4, 2, 3} множества {1, 2, 3, 4}.

- *Подмножества.*

  Каждое множество элементов {1, ..., *n*} содержит подмножество {1, ..., *n* – 1}, являющееся результатом удаления элемента *n*, если такой имеется.

- *Деревья.*

  Что мы получим, удалив корень дерева? Правильно, коллекцию меньших деревьев. А что мы получим, удалив какой-либо лист дерева? Немного меньшее дерево.

- *Графы.*

  Удалите любую вершину графа, и вы получите меньший граф. Теперь разобьем вершины графа на две группы, правую и левую. Разрезав все ребра, соединяющие эти группы, мы получим два меньших графа и набор разорванных ребер.

- *Точки.*

  Возьмем облако точек и разобьем его линией на две группы. Теперь у нас есть два меньших облака точек.

- *Многоугольники.*

  Соединив хордой две несмежные вершины простого многоугольника с *n* вершинами, мы получим два меньших многоугольника.

- *Строки.*

Что мы получим, удалив первый символ строки? Другую, более короткую строку[2].

Для рекурсивного описания объектов требуются как правила разложения, так и *базовые объекты*, а именно спецификации простейшего объекта, далее которого разложение не идет. Такие базовые объекты обычно легко определить. Перестановки и подмножества нулевого количества объектов обозначаются как {}. Наименьшее дерево или граф состоят из одной вершины, а наименьшее облако точек состоит из одной точки. С многоугольниками немного посложнее — наименьшим многоугольником является треугольник. Наконец, пустая строка содержит нулевое количество знаков. Решение о том, содержит ли базовый объект нулевое количество элементов или один элемент, является, скорее, вопросом вкуса и удобства, нежели какого-либо фундаментального принципа.

Приведенные здесь рекурсивные разложения применяются для определения многих алгоритмов, рассматриваемых в этой книге. Обращайте на них внимание.

## 1.5. Доказательство от противного

Хотя некоторые специалисты в компьютерной области умеют доказывать только методом индукции, это утверждение не относится ко всем им. Иногда наилучшим подходом будет использовать метод *доказательства от противного (обратного)*.

Базовая процедура применения этого подхода следующая:

1. Предполагаем, что та или иная гипотеза (утверждение, которое требуется доказать) является *ложной*.
2. Разрабатываем определенные логические последствия этого предположения.
3. Показываем, что одно последствие является очевидно ложным, тем самым демонстрируя ложность нашего предположения и правильность гипотезы.

Классическим примером доказательства от противного является евклидово доказательство существования бесконечного количества простых чисел, т. е. целых чисел $n$, таких как 2, 3, 5, 7, 11, ..., которые не имеют нетривиальных множителей, а только 1 и само $n$. Отрицание этого утверждения может состоять в том, что существует только конечное количество простых чисел, скажем, $m$, которые можно перечислить, как $p_1$, ..., $p_m$. Предположим, что так оно и есть, и будем работать с этим случаем.

Простые числа обладают особыми свойствами деления. Создадим целое число $N$, являющееся произведением «всех» перечисленных простых чисел:

$$N = \prod_{i=1}^{m} p_i$$

Это целое число $N$ обладает свойством деления без остатка на все известные простые числа вследствие особенности его создания.

---

[2] Внимательный читатель предыдущего издания подметил, что салаты являются рекурсивными объектами, тогда как гамбургеры нет. Съев ложку салата (или убрав из него один ингредиент), получим салат, только меньшего размера. А откусив кусок гамбургера, получим нечто отвратительное.

Но рассмотрим целое число $N + 1$. Оно не может делиться на число $p_1 = 2$, поскольку на него делится $N$. То же самое относится и к числу $p_2 = 3$ и всем другим перечисленным простым числам. Поскольку число $N + 1$ не обладает никакими нетривиальными множителями, это означает, что оно должно быть простым числом. Но мы же утверждали, что существует точно $m$ простых чисел, ни одно из которых не является $N + 1$. Это утверждение оказалось ложным, поэтому количество простых чисел не может быть ограниченным. Туше!

Чтобы доказательство от противного было убедительным, конечное последствие должно быть явно, до абсурда ложным. Нечеткие результаты неубедительны. Также важно, чтобы это противоречие было логическим последствием исходного предположения.

Доказательства от противного используются в алгоритмах для работы с минимальными остовными деревьями в *разд. 8.1*.

## 1.6. Истории из жизни

Самый лучший способ узнать, какое огромное влияние тщательная разработка алгоритма может иметь на производительность, — это ознакомиться с примерами из реальной жизни. Внимательно изучая опыт других людей, мы учимся использовать этот опыт в нашей собственной работе.

В различных местах этой книги приводится несколько рассказов об успешном (а иногда и неуспешном) опыте нашей команды в разработке алгоритмов решения реальных задач. Я надеюсь, что вы сможете перенять и усвоить опыт и знания, полученные нами в процессе этих разработок, чтобы использовать их в качестве моделей для ваших собственных решений.

*Все, изложенное в этих рассказах, действительно произошло.* Конечно же, в изложении истории слегка приукрашены, и диалоги в них подредактированы. Но я приложил все усилия, чтобы правдиво описать весь процесс, начиная от постановки задачи до выдачи решения, чтобы вы могли проследить его в действии.

В своих рассказах я пытаюсь уловить образ мышления алгоритмиста в процессе решения задачи.

Эти истории из жизни обычно затрагивают, по крайней мере, одну задачу из каталога *второй части* книги. Когда такая задача встречается, дается ссылка на соответствующий раздел каталога. Таким образом подчеркиваются достоинства моделирования разрабатываемого приложения в терминах стандартных алгоритмических задач. Пользуясь каталогом задач, вы сможете в любое время получить известную информацию о решаемой задаче.

## 1.7. История из жизни. Моделирование проблемы ясновидения

Я занимался обычными делами, когда на меня нежданно-негаданно свалился этот запрос в виде телефонного звонка.

— Профессор Скиена, я надеюсь, что Вы сможете помочь мне. Я — президент компании Lotto System Group, Inc., и нам нужен алгоритм решения проблемы, возникающей в нашем последнем продукте.

— Буду рад помочь Вам, — ответил я. В конце концов, декан моего инженерного факультета всегда призывает преподавательский состав к более широкому взаимодействию с предпринимателями.

— Наша компания продает программу, предназначенную для развития способностей наших клиентов к ясновидению и предсказанию выигрышных лотерейных номеров[3]. Стандартный лотерейный билет содержит шесть номеров, выбираемых из большего количества последовательных номеров, например, от 1 до 44. Но после соответствующей тренировки наши клиенты могут мысленно увидеть, скажем, 15 номеров из 44 возможных, по крайней мере, четыре из которых будут на выигрышном билете. Вы все пока понимаете?

— Скорее нет, чем да, — сказал я, но тут снова вспомнил, что наш декан призывает нас взаимодействовать с представителями бизнес-сообщества.

— Наша проблема заключается в следующем. После того как ясновидец сузил выбор номеров от 44 до 15, из которых он уверен в правильности, по крайней мере, четырех, нам нужно найти эффективный способ применить эту информацию. Допустим, угадавшему, по крайней мере, три правильных номера выдается денежный приз. Нам нужен алгоритм, чтобы составить наименьший набор билетов, которые нужно купить, чтобы гарантировать выигрыш, по крайней мере, одного приза.

— При условии, что ясновидец не ошибается?

— Да, при условии, что ясновидец не ошибается. Нам нужна программа, которая распечатывает список всех возможных комбинаций выигрышных номеров билетов, которые он должен купить с минимальными затратами. Можете ли вы помочь нам с решением этой задачи?

Возможно, они и в самом деле были ясновидцами, т. к. обратились как раз туда, куда надо. Определение наилучшего подмножества номеров билетов попадает в разряд комбинаторных задач. А точнее, это некий тип задачи о покрытии множества, в которой каждый покупаемый билет «покрывает» некоторые из возможных четырехэлементных подмножеств увиденного ясновидцем пятнадцатиэлементного множества. Определение наименьшего набора билетов для покрытия всех возможностей представляет собой особый экземпляр NP-полной задачи о *покрытии множества* (рассматривается в *разд. 21.1*), считающейся вычислительно неразрешимой[4].

Поставленная задача действительно была особым экземпляром задачи о покрытии множества, полностью определяемого всего лишь четырьмя числами: размером $n$ возможного множества $S$ ($n \approx 15$), количеством номеров $k$ на каждом билете ($k \approx 6$), количеством обещаемых ясновидцем выигрышных номеров $j$ из множества $S$ ($j = 4$) и, нако-

---

[3] Это реальный случай.

[4] NP-полная задача (NP-complete problem) — вычислительная задача называется *NP-полной* (от *англ.* nondeterministic polynomial — недетерминированная с полиномиальным временем), если для нее не существует эффективных алгоритмов решения (см. *главу 11*). — *Прим. ред.*

нец, количеством совпадающих номеров $l$, необходимых, чтобы выиграть приз ($l = 3$). На рис. ЦВ-1.11 показано покрытие экземпляра меньшего размера, где $n = 5$, $k = 3$, $l = 2$, и без помощи ясновидца (означая, что $j = 5$).

— Хотя найти точный минимальный набор билетов с помощью эвристических методов будет трудно, я смогу дать вам покрывающий набор номеров, достаточно близкий к самому дешевому, — ответил я ему. — Вам этого будет достаточно?

— При условии, что ваша программа создает лучший набор номеров, чем программа моего конкурента, это будет то, что надо. Его система не всегда гарантирует выигрыш. Очень признателен за вашу помощь, профессор Скиена.

— Один вопрос напоследок. Если с помощью вашей программы люди могут натренироваться выбирать выигрышные лотерейные билеты, то почему вы сами не пользуетесь ею, чтобы выигрывать в лотерею?

— Надеюсь встретиться с Вами в ближайшее время, профессор Скиена. Благодарю за помощь.

Я повесил трубку и начал обдумывать дальнейшие действия. Задача выглядела как идеальный проект для какого-либо смышленого студента. После моделирования задачи посредством множеств и подмножеств основные компоненты решения выглядели достаточно просто:

- нам была нужна возможность генерировать все подмножества $k$ номеров из потенциального множества $S$. Алгоритмы генерирования и ранжирования подмножеств представлены в *разд. 17.5*;
- нам была нужна правильная формулировка того, что именно означает требование иметь покрывающее множество приобретенных билетов. Очевидным критерием, что требование выполнено, мог быть наименьший набор билетов, включающий, по крайней мере, один билет, содержащий каждое из $\binom{n}{l}$ $l$-подмножеств множества $S$, за которое может выдаваться приз;
- нам нужно было отслеживать уже рассмотренные призовые комбинации. Такие комбинации номеров билетов понадобятся, чтобы покрыть как можно больше еще не охваченных призовых комбинаций с учетом того, что текущие охваченные комбинации являются подмножеством всех возможных комбинаций. Структуры данных для подмножеств рассматриваются в *разд. 15.5*. Наилучшим кандидатом выглядел вектор битов, который бы за постоянное время давал ответ, охвачена ли уже та или иная комбинация;
- нужен был и механизм поиска, чтобы решить, какой следующий билет покупать. Для небольших множеств можно было проверить все возможные подмножества комбинаций и выбрать из них самую меньшую. Для множеств большего размера выигрышные комбинации номеров билетов для покупки можно было выбирать с помощью какого-либо процесса рандомизированного поиска наподобие имитации отжига (см. *разд. 12.6.3*), чтобы покрыть как можно больше комбинаций. Выполняя эту рандомизированную процедуру несколько раз и выбирая самые лучшие решения, мы смогли бы, скорее всего, получить хороший набор комбинаций номеров билетов.

Способный студент Фаяз Юнас (Fayyaz Younas) принял вызов реализовать алгоритм. На основе указаного перечня задач он реализовал алгоритм поиска методом полного перебора и смог найти оптимальные решения задач, в которых $n \leq 5$. Для решения задачи с большим значением $n$ он реализовал процедуру рандомизированного поиска, которую отлаживал, пока не остановился на самом лучшем варианте. Наконец наступил день, когда мы могли позвонить в компанию Lotto Systems Group и сказать им, что мы решили задачу.

— Результат работы нашей программы таков: оптимальным решением для $n = 15$, $k = 6, j = 4, l = 3$ будет покупка 28 билетов.

— Двадцать восемь билетов! — выразил недовольство президент. — В вашей программе, должно быть, есть ошибка. Вот пять билетов, которых будет достаточно, чтобы покрыть все варианты *дважды*: {2, 4, 8, 10, 13, 14}, {4, 5, 7, 8, 12, 15}, {1, 2, 3, 6, 11, 13}, {3, 5, 6, 9, 10, 15}, {1, 7, 9, 11, 12, 14}.

Мы повозились с этим примером немного и должны были признать, что он был прав. *Мы неправильно смоделировали задачу!* В действительности, нам не нужно было покрывать явно все возможные выигрышные комбинации. Объяснение представлено на рис. ЦВ-1.12 в виде двухбилетного решения нашего предыдущего четырехбилетного примера.

Хотя пары {2, 4}, {2, 5}, {3, 4} и {3, 5} и не присутствуют явно в одном из двух наших билетов, эти пары, плюс любой возможный номер третьего билета, должны создать пару или в {1, 2, 3}, или в {1, 4, 5}. Мы пытались покрыть слишком много комбинаций, а дрожащие над каждой копейкой ясновидцы не желали платить за такую расточительность.

К счастью, у этой истории хороший конец. Общий принцип нашего поискового решения оставался применимым для реальной задачи. Все, что нужно было сделать, так это исправить, какие подмножества засчитываются за покрытие рассчитанным набором комбинаций номеров билетов. Сделав это исправление, мы получили требуемые результаты. В компании Lotto Systems Group с благодарностью приняли нашу программу для внедрения в свой продукт. Будем надеяться, что они сорвут большой куш с ее помощью.

Мораль этой истории заключается в том, что необходимо удостовериться в правильности моделирования задачи, прежде чем пытаться решить ее. В нашем случае мы разработали правильную модель, но недостаточно глубоко проверили ее, перед тем, как начинать создавать программу на ее основе. Наше заблуждение было бы быстро выявлено, если бы, прежде чем приступать к решению реальной проблемы, мы проработали небольшой пример вручную и обсудили результаты с нашим клиентом. Но мы смогли выйти из этой ситуации с минимальными отрицательными последствиями благодаря правильности нашей первоначальной формулировки и использованию четко определенных абстракций для таких задач, как генерирование $k$-элементных подмножеств методом ранжирования, структуры данных множества и комбинаторного поиска.

# 1.8. Прикидка

Когда не удается получить точный ответ, лучше всего будет предположить его. Предположение на основе определенных правил называется *прикидкой*. Способность выполнять предварительную оценку разных величин — например, времени исполнения программы, является ценным навыком в области разработки алгоритмов, как и в любом техническом предприятии.

Задачи приблизительной оценки лучше всего решаются с помощью какого-либо процесса логических рассуждений — обычно комбинации методических вычислений и аналогий. *Методические вычисления* предоставляют ответ в виде функции величин, которые или *а*) уже известны, или *б*) можно найти в сети, или *в*) можно предположить с достаточной степенью уверенности. *Аналогии* ссылаются на прошлые события, выбирая из них те, которые кажутся похожими на некоторый аспект решаемой задачи.

Однажды я дал задачу своей группе приблизительно определить количество центов в большой стеклянной банке, и получил ответы в диапазоне от 250 до 15 000. Если провести правильные аналогии, то оба эти значения выглядят весьма нелепыми:

- ролик центов размером, приблизительно равным размеру самого большого пальца, содержит 50 монет. Пять таких роликов могут свободно поместиться в руке, не требуя большой банки;
- пятнадцать тысяч центов эквивалентны сумме в $150. Мне никогда не удалось насобирать такую сумму даже в большой куче разных монет, а здесь мы говорим только о центах!

Но среднее значение приблизительных оценок класса оказалось очень близким к правильному значению — 1879. Лично мне на ум приходят, по крайней мере, три методических способа приблизительно оценить количество центовых монет в банке:

- *По объему.*

  Банка диметром около 5 дюймов (12,7 см) была заполнена монетами до уровня высотой приблизительно в десять монет, поставленных ребром друг на друга. Диаметр центовой монеты приблизительно в десять раз больше ее толщины. Радиус нижнего слоя центов в банке составлял приблизительно пять монет. Таким образом:

$$(10 \times 10) \times (\pi \times 2.52) \approx 1962.5.$$

- *По весу.*

  По весу банка ощущалась как шар для боулинга. Умножив количество центовых монет в одном фунте (181 согласно полученным в результате поиска данным) на вес шара (10 фунтов), получим довольно близкое прикидочное значение в 1810.

- *Используя аналогию.*

  Банка была заполнена монетами на высоту около 8 дюймов, что вдвое больше длины ролика этих монет. Я полагаю, что в банку можно было поместить два вертикальных слоя роликов по десять роликов в каждом, или приблизительно 1000 монет.

Самая лучшая практика для прикидок — попытаться решить задачу несколькими способами и посмотреть, насколько полученные результаты близки по величине. В этом

примере все прикидочные результаты не отличаются друг от друга больше чем в два раза, придавая мне уверенность в достаточной правильности моего ответа.

Попробуйте выполнить некоторые упражнения по прикидке, приведенные в конце этой главы, применяя несколько разных подходов. При правильно выбранных подходах наименьшее и наибольшее прикидочные значения не должны отличаться друг от друга больше чем в два-десять раз, в зависимости от сути задачи. Правильный процесс логических рассуждений имеет намного большую важность, чем полученные фактические результаты.

## Замечания к главе

В каждой хорошей книге, посвященной алгоритмам, отражается подход ее автора к их разработке. Тем, кто хочет ознакомиться с альтернативными точками зрения, особенно рекомендуется прочитать книги [CLRS09], [KT06], [Man89] и [Rou17].

Формальные доказательства правильности алгоритма являются важными и заслуживают более полного рассмотрения, чем можно предоставить в этой главе. Методы верификации программ обсуждаются в книге [Gri89].

Задача календарного планирования ролей в фильмах является особым случаем общей задачи *независимого множества*, которая рассматривается в *разд. 19.2*. В качестве входных экземпляров допускаются только интервальные графы, в которых вершины графа $G$ можно представить линейными интервалами, а $(i, j)$ является ребром $G$ тогда и только тогда, когда интервалы пересекаются. Этот интересный и важный класс графов подробно рассматривается в книге [Gol04].

Колонка «Программистские перлы» Джона Бентли (Jon Bentley) является, наверное, самой известной коллекцией историй из жизни разработчиков алгоритмов. Материалы этой колонки были изданы в двух книгах: [Ben90] и [Ben99]. Еще одна прекрасная коллекция историй из жизни собрана в книге [Bro95]. Хотя эти истории имеют явный уклон в сторону разработки и проектирования программного обеспечения, они также представляют собой кладезь мудрости. Все программисты должны прочитать эти книги, чтобы получить и пользу, и удовольствие.

Наше решение задачи о покрытии множества лотерейных билетов более полно представлено в книге [YS96].

## 1.9. Упражнения

### Поиск контрпримеров

1. *[3]*[5] Докажите, что значение $a + b$ может быть меньшим, чем значение $\min(a, b)$.
2. *[3]* Докажите, что значение $a \times b$ может быть меньшим, чем значение $\min(a, b)$.
3. *[5]* Начертите сеть дорог с двумя точками $a$ и $b$ такими, что маршрут между ними, преодолеваемый за кратчайшее время, не является самым коротким.

[5] Цифрами в квадратных скобках отмечен уровень сложности упражнений в диапазоне от *[1]* до *[10]*. — *Прим. ред.*

4. *[5]* Начертите сеть дорог с двумя точками $a$ и $b$, самый короткий маршрут между которыми не является маршрутом с наименьшим количеством поворотов.

5. *[4]* Задача о рюкзаке: имея множество целых чисел $S = \{s_1, s_2, ..., s_n\}$ и целевое число $T$, найти такое подмножество множества $S$, сумма которого в точности равна $T$. Например, множество $S = \{1, 2, 5, 9, 10\}$ содержит такое подмножество, сумма элементов которого равна $T = 22$, но не $T = 23$.

   Найти контрпримеры для каждого из следующих алгоритмов решения задачи о рюкзаке, т. е. найти такое множество $S$ и число $T$, при которых алгоритм не находит решения, полностью заполняющего рюкзак, хотя такое решение и существует:

   - вкладывать элементы множества $S$ в рюкзак в порядке слева направо, если они подходят (т. е. алгоритм «первый подходящий»);
   - вкладывать элементы множества $S$ в рюкзак в порядке от наименьшего до наибольшего (т. е. используя алгоритм «первый лучший»);
   - вкладывать элементы множества $S$ в рюкзак в порядке от наибольшего до наименьшего.

6. *[5]* Задача о покрытии множества: имея множество $S$ семейства подмножеств $S_1, ..., S_m$ универсального множества $U = \{1, ..., n\}$, найдите семейство подмножеств $T \subseteq S$ наименьшей мощности, чтобы $\bigcup_{t_i} \in T^{t_i} = U$. Например, для семейства подмножеств $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{1, 4\}$ и $S_4 = \{2, 5\}$ покрытием множества $\{1, ..., 5\}$ будет семейство подмножеств $S_1$ и $S_2$. Приведите контрпример для следующего алгоритма: выбираем самое мощное подмножество для покрытия, после чего удаляем все его элементы из универсального множества. Повторяем добавление подмножества, содержащего наибольшее количество неохваченных элементов, пока все элементы не будут покрыты.

7. *[5]* Задача максимальной клики графа $G = (V, E)$ требует найти такое наибольшее подмножество $C$ вершин $V$, в котором между каждой парой вершин в $C$ есть ребро в $E$. Создайте контрпример для следующего алгоритма: сортируем вершины графа $G$ по степени в нисходящем порядке. Рассматриваем все вершины в порядке степени, добавляя вершину в клику, если она является соседом всех вершин — членов клики в текущий момент. Повторяем процесс до тех пор, пока не будут рассмотрены все вершины.

## Доказательство правильности

8. *[3]* Докажите правильность следующего рекурсивного алгоритма умножения двух натуральных чисел для всех целочисленных констант $c \geq 2$:

```
Multiply(y, z)
            if z = 0 then return(0) else
            return(Multiply(cy,[z/c]) + y·(z mod c))
```

9. *[3]* Докажите правильность следующего алгоритма вычисления полинома $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$:

```
Horner(A, x)
  p = a_n
  for i from n-1 to 0
    p = p · x + a_i
  return p
```

10. *[3]* Докажите правильность следующего алгоритма сортировки:

```
Bubblesort (A)
  var int i,j
  for i from n to 1
    for j from 1 to i - 1
      if (A[j]>A[j+1]
        меняем местами значения A[j] и A[j + 1]
```

11. *[5]* Наибольшим общим делителем (нод) положительных целых чисел $x$ и $y$ является наибольшее такое целое число $d$, на которое делится как число $x$, так и число $y$. Евклидов алгоритм для вычисления нод($x$, $y$), где $x > y$, сводит исходную задачу к более простой:

$$\text{нод}(x, y) = \text{нод}(y, x \bmod y)$$

Докажите правильность евклидова алгоритма.

## Математическая индукция

Для доказательства пользуйтесь методом математической индукции.

12. *[3]* Докажите, что $\sum_{i=1}^{n} i = n(n+1)/2$ для $n \geq 0$.

13. *[3]* Докажите, что $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ для $n \geq 0$.

14. *[3]* Докажите, что $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$ для $n \geq 0$.

15. *[3]* Докажите, что $\sum_{i=1}^{n} i(i+1)(i+2) = n(n+1)(n+2)(n+3)/4$.

16. *[5]* Докажите, что $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ для $n \geq 1$, $a \neq 1$.

17. *[3]* Докажите, что $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$ для $n \geq 1$.

18. *[3]* Докажите, что $n^3 + 2n$ делится на 3 для всех $n \geq 0$.

19. *[3]* Докажите, что дерево с $n$ вершинами имеет в точности $n - 1$ ребер.

20. *[3]* Докажите, что сумма кубов первых $n$ положительных целых чисел равна квадрату суммы этих целых чисел, т. е.

$$\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2 .$$

## Приблизительные подсчеты

21. *[3]* Содержат ли все ваши книги, по крайней мере, миллион страниц? Каково общее количество страниц всех книг в вашей институтской библиотеке?

22. *[3]* Сколько слов содержит эта книга?

23. *[3]* Сколько часов составляет один миллион секунд? А сколько дней? Выполните все необходимые вычисления в уме.

24. *[3]* Сколько городов и поселков в Соединенных Штатах?

25. *[3]* Сколько кубических километров воды изливается из устья Миссисипи каждый день? Не пользуйтесь никакой справочной информацией. Опишите все предположения, сделанные вами для получения ответа.

26. *[3]* Сколько точек Starbucks или McDonald's в вашей стране?

27. *[3]* Сколько времени потребовалось бы, чтобы опорожнить ванну через соломинку для питья холодных напитков?

28. *[3]* В каких единицах измеряется время доступа к жесткому диску: в миллисекундах (тысячных долях секунды) или микросекундах (миллионных долях секунды)? Сколько времени занимает доступ к слову в оперативной памяти вашего компьютера: больше или меньше микросекунды? Сколько инструкций может выполнить центральный процессор вашего компьютера в течение года, если компьютер постоянно держать включенным?

29. *[4]* Алгоритм сортировки выполняет сортировку 1000 элементов за 1 секунду. Сколько времени займет сортировка 10 000 элементов:

    - если время исполнения алгоритма прямо пропорционально $n^2$?
    - если время исполнения алгоритма, по грубым оценкам, пропорционально $n\log n$?

## Проекты по реализации

30. *[5]* Реализуйте два эвристических алгоритма решения задачи коммивояжера из *разд. 1.1*. Какой из них выдает на практике более качественные решения? Можете ли вы предложить эвристический алгоритм, работающий лучше любого из них?

31. *[5]* Опишите способ проверки достаточности покрытия тем или иным множеством комбинаций номеров лотерейных билетов из задачи в *разд. 1.6*. Напишите программу поиска хороших множеств комбинаций номеров билетов.

## Задачи, предлагаемые на собеседовании

32. *[5]* Напишите функцию деления целых чисел, которая не использует ни оператор деления (/), ни оператор умножения (*). Функция должна быть быстродействующей.

33. *[5]* У вас есть 25 лошадей. В каждой скачке может участвовать не больше 5 лошадей. Требуется определить первую, вторую и третью по скорости лошадь. Найдите минимальное количество скачек, позволяющих решить эту задачу.

34. *[3]* Сколько настройщиков пианино во всем мире?

35. *[3]* Сколько бензоколонок в Соединенных Штатах?

36. *[3]* Сколько весит лед на хоккейном поле?

37. *[3]* Сколько километров дорог в Соединенных Штатах?

38. *[3]* Сколько раз в среднем нужно открыть наугад телефонный справочник Манхэттена, чтобы найти определенного человека?

## LeetCode

LeetCode — это веб-сайт, на котором люди могут попрактиковаться в решении задач кодирования и подготовиться к техническим собеседованиям:

1. **https://leetcode.com/problems/daily-temperatures/**
2. **https://leetcode.com/problems/rotate-list/**
3. **https://leetcode.com/problems/wiggle-sort-ii/**

## HackerRank

HackerRank — это социальная платформа, которая предлагает задания разной сложности по программированию:

1. **https://www.hackerrank.com/challenges/array-left-rotation/**
2. **https://www.hackerrank.com/challenges/kangaroo/**
3. **https://www.hackerrank.com/challenges/hackerland-radio-transmitters/**

## Задачи по программированию

Эти задачи доступны на сайте **http://uva.onlinejudge.org**. Так как сайты англоязычные, названия задач даются на исходном языке:

1. «The $3n + 1$ problem», глава 1, задача 100.
2. «The Trip», глава 1, задача 10137.
3. «Australian Voting», глава 1, задача 10142.

ГЛАВА 2

# Анализ алгоритмов

Алгоритмы являются принципиально важным компонентом информатики, поскольку их изучение не требует использования языка программирования или компьютера. Это означает необходимость в методах, позволяющих сравнивать эффективность алгоритмов, не прибегая к их реализации. Самыми значимыми из этих инструментов являются модель вычислений RAM и асимптотический анализ вычислительной сложности.

Для оценки производительности алгоритмов применяется асимптотическая нотация («Big Oh»). Такой способ оценки производительности является наиболее трудным материалом в этой книге. Но когда вы поймете интуитивную подоплеку этого формализма, вам будет намного легче разобраться с ним.

## 2.1. Модель вычислений RAM

Разработка машинно-независимых алгоритмов основывается на гипотетическом компьютере, называемом *машиной с произвольным доступом к памяти* (Random Access Machine), или RAM-машиной. Согласно этой модели наш компьютер работает таким образом:

- для исполнения любой *простой* операции (`+`, `*`, `-`, `=`, `if`, `call`) требуется ровно один временной шаг;
- циклы и подпрограммы не считаются простыми операциями, а состоят из нескольких простых операций. Нет смысла считать подпрограмму сортировки одношаговой операцией, поскольку для сортировки 1 000 000 элементов потребуется определенно намного больше времени, чем для сортировки десяти элементов. Время исполнения цикла или подпрограммы зависит от количества итераций или специфического характера подпрограммы;
- каждое обращение к памяти занимает один временной шаг. Кроме этого, наш компьютер обладает неограниченным объемом оперативной памяти. Кэш и диск в модели RAM не применяются.

Время исполнения алгоритма в RAM-модели вычисляется по общему количеству шагов, требуемых алгоритму для решения того или иного экземпляра задачи. Допуская, что наша RAM-машина исполняет определенное количество шагов/операций за секунду, количество шагов легко перевести в единицы времени.

Может показаться, что RAM-модель является слишком упрощенным представлением работы компьютеров. В конце концов, на большинстве процессоров умножение двух чисел занимает больше времени, чем сложение, что не вписывается в первое предположение модели. Второе предположение может быть нарушено удачной оптимизацией

цикла компилятором или гиперпотоковыми возможностями процессора. Наконец, время обращения к данным может значительно разниться в зависимости от расположения данных в иерархии памяти. Таким образом, по сравнению с настоящим компьютером все три основные допущения для RAM-машины неверны.

Тем не менее, несмотря на эти несоответствия настоящему компьютеру, RAM-модель является *превосходной* моделью для понимания того, как алгоритм будет работать на настоящем компьютере. Она обеспечивает хороший компромисс, отражая поведение компьютеров и одновременно являясь простой в использовании. Эти характеристики делают RAM-модель полезной для практического применения.

Любая научная модель полезна лишь в определенных рамках. Возьмем, например, модель плоской Земли. Можно спорить, что это неправильная модель, т. к. еще древние греки знали, что в действительности Земля круглая. Но модель плоской Земли достаточно точна для закладки фундамента дома. Более того, при этом с моделью плоской Земли настолько удобнее работать, что использование модели сферической Земли[1] для этой цели даже не приходит в голову.

Та же самая ситуация наблюдается и в случае с RAM-моделью вычислений — мы создаем, вообще говоря, очень полезную абстракцию. Довольно трудно создать алгоритм, для которого RAM-модель выдаст существенно неверные результаты. Устойчивость этой модели позволяет анализировать алгоритмы машинно-независимым способом.

#### Подведение итогов

Алгоритмы можно изучать и анализировать, не прибегая к использованию конкретного языка программирования или компьютерной платформы.

## 2.1.1. Анализ сложности наилучшего, наихудшего и среднего случая

С помощью RAM-модели можно подсчитать количество шагов, требуемых алгоритму для исполнения любого экземпляра задачи. Но чтобы получить общее представление о том, насколько хорошим или плохим является алгоритм, нам нужно знать, как он работает со *всеми* возможными экземплярами задачи.

Чтобы понять, что означает наилучший, наихудший и средний случай сложности алгоритма (т. е. время его исполнения в соответствующем случае), нужно рассмотреть исполнение алгоритма на всех возможных экземплярах входных данных. В случае задачи сортировки множество входных экземпляров состоит из всех $n$ возможных компоновок ключей для всех возможных значений $n$. Каждый входной экземпляр можно представить в виде точки графика (рис. 2.1), где ось $x$ представляет размер входа задачи (для сортировки это будет количество элементов, подлежащих сортировке), а ось $y$ — количество шагов, требуемых алгоритму для обработки такого входного экземпляра.

Эти точки естественным образом выстраиваются столбцами, т. к. размер входа может быть только целым числом (т. е. сортировка 10,57 элемента лишена смысла). На графике этих точек можно определить три представляющие интерес функции:

---

[1] В действительности Земля не совсем сферическая, но модель сферической Земли удобна для работы с такими понятиями, как широта и долгота.

- *сложность алгоритма в наихудшем случае* — это функция, определяемая максимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n*. Этот случай отображается кривой, проходящей через самую высшую точку каждого столбца;
- *сложность алгоритма в наилучшем случае* — это функция, определяемая минимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n*. Этот случай отображается кривой, проходящей через самую низшую точку каждого столбца;
- *сложность алгоритма в среднем случае*, или его *ожидаемое время*, — это функция, определяемая средним количеством шагов, требуемых для обработки всех экземпляров размером *n*.

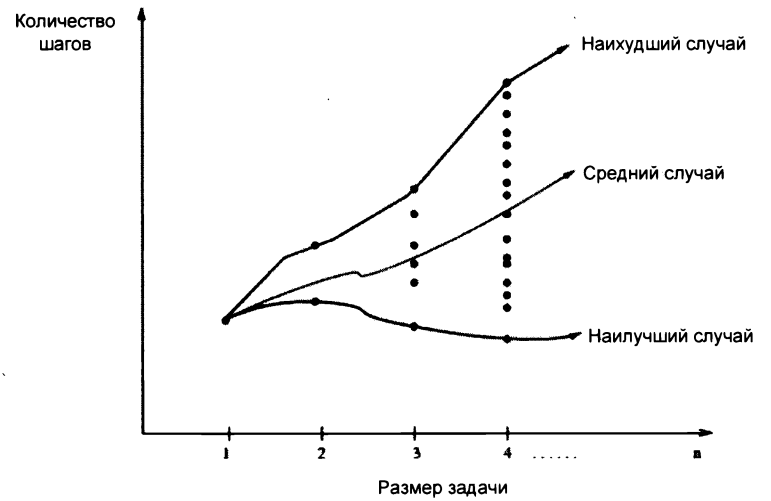



**Рис. 2.1.** Наилучший, наихудший и средний случаи сложности алгоритма

На практике обычно наиболее важной является оценка сложности алгоритма в наихудшем случае. Для многих людей это противоречит здравому смыслу. Рассмотрим пример. Вы собираетесь посетить казино, имея в кармане *n* долларов. Каковы возможные исходы? В наилучшем случае вы выиграете само казино, но хотя такое развитие событий и, возможно, вероятность его настолько ничтожна, что даже не стоит думать об этом. В наихудшем случае вы проиграете все свои *n* долларов — вероятность такого события удручающе высока, и ее легко вычислить. Средний случай, когда типичный игрок проигрывает в казино 87,32% взятых с собой денег, трудно рассчитать и даже определить. Что именно означает *средний*? Глупые люди проигрывают больше, чем умные, так вы умнее или глупее, чем средний человек, и насколько? Игроки в очко, которые умеют отслеживать сданные карты, в среднем выигрывают больше, чем игроки, которые не отказываются от нескольких бесплатных рюмок алкогольного напитка, регулярно предлагаемых симпатичными официантками. Чтобы избежать всех этих усложнений и получить самый полезный результат, мы и рассматриваем только наихудший случай.

Сказав все это, анализ времени исполнения среднего случая окажется очень важным для *рандомизованных алгоритмов*, т. е. алгоритмов, использующих случайные числа для принятия решений. Если в рулетке сделать $n$ независимых ставок по одному доллару на красное или черное, то величина ожидаемых потерь на самом деле будет вполне определенной, равняясь $(2n / 38)$ долларов, т. к. в американских казино рулеточное колесо разделено на 18 красных, 18 черных и два проигрышных зеленых сектора: 0 и 00.

### Подведение итогов

Каждая из этих временны́х сложностей определяет числовую функцию для любого алгоритма, соотносящую время исполнения с размером задачи. Эти функции определены так же строго, как и любые другие числовые функции, будь то уравнение $y = x^2 - 2x + 1$ или цена акций компании Alphabet в зависимости от времени. Но функции временной сложности настолько трудны для понимания, что, прежде чем приступать к их анализу, их нужно упростить, используя для этого асимптотическую нотацию — обозначение «Big Oh» («*O*-большое»).

## 2.2. Асимптотические («Big Oh») обозначения

Временну́ю сложность наилучшего, наихудшего и среднего случаев для любого алгоритма можно представить как числовую функцию от размеров возможных экземпляров задачи. Но работать с этими функциями очень трудно, поскольку они обладают следующими свойствами:

- *являются слишком волнистыми.*

  Время исполнения алгоритма, например, двоичного поиска, обычно меньше для массивов, имеющих размер $n = 2^k - 1$, где $k$ — целое число. Эта особенность не имеет большого значения, но служит предупреждением, что *точная* функция временной сложности любого алгоритма вполне может иметь неровный график с большим количеством небольших выпуклостей и впадин, как показано на рис. 2.2;

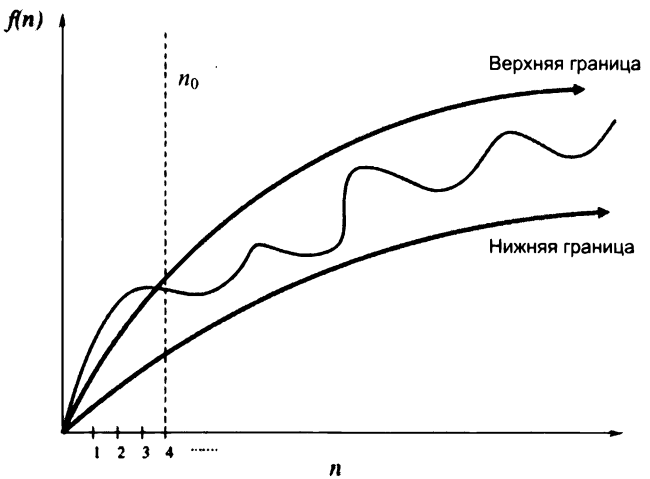



**Рис. 2.2.** Верхняя и нижняя границы, действительные для $n > n_0$, сглаживают волнистость сложных функций

- *требуют слишком много информации для точного определения.*

  Чтобы подсчитать точное количество инструкций RAM-машины, исполняемых в худшем случае, нужно, чтобы алгоритм был расписан в подробностях полной компьютерной программы. Более того, точность ответа зависит от маловажных деталей кодировки (например, был ли употреблен в коде оператор `case` вместо вложенных операторов `if`). Точный анализ наихудшего случая — например, такого:

$$T(n) = 12\,754n^2 + 4353n + 834\lg_2 n + 13\,546,$$

  очевидно, был бы очень трудной задачей, решение которой не предоставляет нам никакой дополнительной информации, кроме той, что «с увеличением $n$ временная сложность возрастает квадратически».

Оказывается, намного легче работать с верхней и нижней границами функций временной сложности, используя для этого асимптотические («Big Oh») обозначения. Асимптотические обозначения позволяют упростить анализ, поскольку игнорируют детали, которые не влияют на сравнение эффективности алгоритмов.

В частности, в асимптотических обозначениях игнорируется разница между постоянными множителями. Например, в анализе с применением асимптотического обозначения функции $f(n) = 2n$ и $g(n) = n$ являются идентичными. Это вполне логично в нашей ситуации. Допустим, что определенный алгоритм на языке C исполняется вдвое быстрее, чем тот же алгоритм на языке Java. Этот постоянный множитель, равняющийся двум, не может предоставить нам никакой информации о собственно алгоритме, т. к. в обоих случаях исполняется один и тот же алгоритм. При сравнении алгоритмов такие постоянные коэффициенты должны игнорироваться.

Формальные определения, связанные с асимптотическими обозначениями, выглядят таким образом:

- $f(n) = O(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c \cdot g(n)$. Иными словами, существует такая константа $c$, при которой $f(n) \leq c \cdot g(n)$ для каждого достаточно большого значения $n$ (т. е. для всех $n \geq n_0$ для некоторой константы $n_0$). Этот случай показан на рис. 2.3, *а*;
- $f(n) = \Omega(g(n))$ означает, что функция $f(n)$ *ограничена снизу* функцией $c \cdot g(n)$. Иными словами, существует такая константа $c$, для которой $f(n) \geq c \cdot g(n)$ для всех $n \geq n_0$. Этот случай показан на рис. 2.3, *б*;

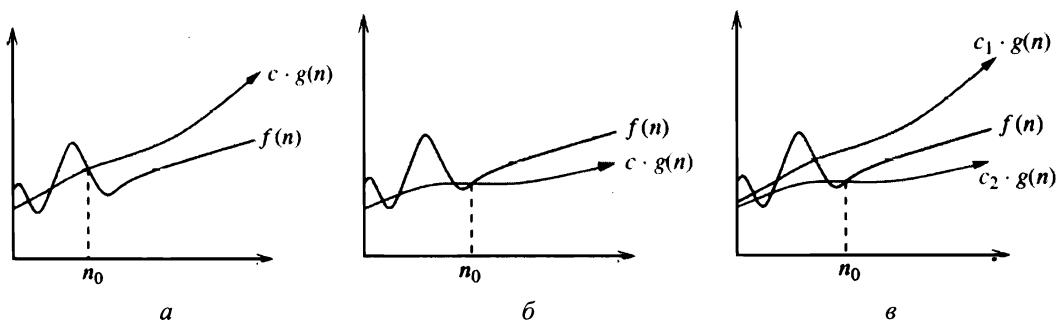



**Рис. 2.3.** Графическая иллюстрация асимптотических обозначений:
*а* — $f(n) = O(g(n))$; *б* — $f(n) = \Omega(g(n))$; *в* — $f(n) = \Theta(g(n))$

- $f(n) = \Theta(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c_1 \cdot g(n)$, а *снизу* функцией $c_2 \cdot g(n)$ для всех $n \geq n_0$. Иными словами, существуют константы $c_1$ и $c_2$, для которых $f(n) \leq c_1 \cdot g(n)$ и $f(n) \geq c_2 \cdot g(n)$ для всех $n \geq n_0$. Этот случай показан на рис. 2.3, *в*.

Следовательно, функция $g(n)$ дает нам хорошие ограничения для функции $f(n)$.

В каждом из приведенных определений фигурирует константа $n_0$, после которой эти определения верны. Нас не интересуют небольшие значения $n$, т. е. любые значения слева от $n_0$. В конце концов, нам безразлично, что один алгоритм может отсортировать, скажем, шесть или восемь элементов быстрее, чем другой, но нам нужно знать, какой алгоритм отсортирует быстрее 10 000 или 1 000 000 элементов. В этом отношении асимптотические обозначения позволяют нам игнорировать несущественные детали и концентрироваться на общей картине.

#### Подведение итогов

Анализ наихудшего случая и асимптотические обозначения являются инструментами, которые существенно упрощают задачу сравнения эффективности алгоритмов.

Обязательно убедитесь, что вы понимаете смысл асимптотических обозначений, изучив следующие далее примеры. Конкретные значения констант $c$ и $n_0$ были выбраны потому, что они хорошо иллюстрируют ситуацию, но можно использовать и другие значения этих констант с точно таким же результатом. Вы можете выбрать любые значения констант, при которых сохраняется первоначальное неравенство, — в идеале, следует выбирать такие значения, при которых очевидно, что неравенство соблюдается:

$f(n) = 3n^2 - 100n + 6 = O(n^2)$, т. к. для $c = 3$, $3n^2 > f(n)$;

$f(n) = 3n^2 - 100n + 6 = O(n^3)$, т. к. для с = 1, $n^3 > f(n)$ при $n > 3$;

$f(n) = 3n^2 - 100n + 6 \neq O(n)$, т. к. для любого значения $c > 0$, $cn < f(n)$ при $n > (c + 100)/3$, поскольку $n > (c + 100)/3 \Rightarrow 3n > c + 100 \Rightarrow 3n^2 > cn + 100n > cn + 100n - 6 \Rightarrow 3n^2 - 100n + 6 = f(n) > cn$;

$f(n) = 3n^2 - 100n + 6 = \Omega(n^2)$, т. к. для $c = 2$, $2n^2 < f(n)$ при $n > 100$;

$f(n) = 3n^2 - 100n + 6 \neq \Omega(n^3)$, т. к. для любого значения $c > 0$, $f(n) < c \cdot n^3$ при $n > 3/c + 3$;

$f(n) = 3n^2 - 100n + 6 = \Omega(n)$, т. к. для любого значения с > 0, $f(n) < 3n^2 + 6n^2 = 9n^2$, что $< cn^3$ при $n > \max(9/c, 1)$;

$f(n) = 3n^2 - 100n + 6 = \Theta(n^2)$, т. к. применимо как $O$, так и $\Omega$ ;

$f(n) = 3n^2 - 100n + 6 \neq \Theta(n^3)$, т. к. применимо только $O$;

$f(n) = 3n^2 - 100n + 6 \neq \Theta(n)$, т. к. применимо только $\Omega$.

Асимптотические обозначения позволяют получить приблизительное представление о равенстве функций при их сравнении. Выражение типа $n^2 = O(n^3)$ может выглядеть странно, но его значение всегда можно уточнить, пересмотрев его определение в терминах верхней и нижней границ. Возможно, это обозначение будет более понятным, если здесь рассматривать символ равенства (=) как означающий «одна из функций,

принадлежащих к множеству функций». Очевидно, что $n^2$ является одной из функций, принадлежащих множеству функций $O(n^3)$.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Возвращение к определениям

**ЗАДАЧА.** Верно ли равенство $2^{n+1} = \Theta(2^n)$?

**РЕШЕНИЕ.** Для разработки оригинальных алгоритмов требуется умение и вдохновение. Но при использовании асимптотических обозначений лучше всего подавить все свои творческие инстинкты. Все задачи по асимптотическим обозначениям можно правильно решить, работая с первоначальными определениями.

- Верно ли равенство $2^{n+1} = O(2^n)$? Очевидно, что $f(n) = O(g(n))$ тогда и только тогда, когда существует такая константа $c$, при которой для всех достаточно больших значений $n$ функция $f(n) \le c \cdot g(n)$. Существует ли такая константа? Да, существует, т. к. $2^{n+1} = 2 \cdot 2^n$, и очевидно, что $2 \cdot 2^n \le c \cdot 2^n$ для всех $c \ge 2$.
- Верно ли равенство $2^{n+1} = \Omega(2^n)$? Согласно определению $f(n) = \Omega(g(n))$ тогда и только тогда, когда существует такая константа $c > 0$, при которой для всех достаточно больших значений $n$ функция $f(n) \ge c \cdot g(n)$. Это условие удовлетворяется для любой константы $0 < c \le 2$. Границы $O$-большое и $\Omega$-большое совместно подразумевают $2^{n+1} = \Theta(2^n)$. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Квадраты

**ЗАДАЧА.** Верно ли равенство $(x + y)^2 = O(x^2 + y^2)$?

**РЕШЕНИЕ.** При малейших трудностях в работе с асимптотическим обозначением немедленно возвращаемся к его определению, согласно которому это выражение действительно тогда и только тогда, когда мы можем найти такое значение $c$, при котором $(x + y)^2 \le c(x^2 + y^2)$ для всех достаточно больших значений $x$ и $y$.

Лично я первым делом раскрыл бы скобки в левой части уравнения, т. е. $(x + y)^2 = x^2 + 2xy + y^2$. Если бы в развернутом выражении отсутствовал член $2xy$, то вполне очевидно, что неравенство соблюдалось бы для любого значения $c > 1$. Но поскольку этот член имеется, то нам нужно рассмотреть его связь с выражением $x^2 + y^2$. Если $x \le y$, то $2xy \le 2y^2 \le 2(x^2 + y^2)$. А если $x \ge y$, то $2xy \le 2x^2 \le 2(x^2 + y^2)$. В любом случае теперь мы можем ограничить выражение $2xy$ двойным значением функции с правой стороны $x2 + y2$. Это означает, что $(x + y)^2 \le 3(x^2 + y^2)$, поэтому результат остается в силе. ■

## 2.3. Скорость роста и отношения доминирования

Используя асимптотические обозначения, мы пренебрегаем постоянными множителями, не учитывая их при вычислении функций. При таком подходе функции $f(n) = 0{,}001n^2$ и $g(n) = 1000n^2$ для нас одинаковы, несмотря на то, что значение функции $g(n)$ в миллион раз больше значения функции $f(n)$ для любого $n$.

Причина, по которой достаточно грубого анализа, предоставляемого обозначением $O$-большое, приводится в табл. 2.1, в которой приведены наиболее распространенные функции и их значения для нескольких значений $n$. В частности, здесь можно увидеть

время исполнения $f(n)$ операций алгоритмов на быстродействующем компьютере, выполняющем каждую операцию за одну наносекунду ($10^{-9}$ секунд). На основе представленной в табл. 2.1 информации можно сделать следующие выводы:

- время исполнения всех этих алгоритмов примерно одинаково для значений $n = 10$;
- любой алгоритм с временем исполнения $n!$ становится бесполезным для значений $n \geq 20$;
- диапазон алгоритмов с временем исполнения $2^n$ несколько шире, но они также становятся непрактичными для значений $n > 40$;
- алгоритмы с квадратичным временем исполнения $n^2$ применяются при $n \leq 10\,000$, после чего их производительность начинает резко ухудшаться. Эти алгоритмы, скорее всего, будут бесполезны для значений $n > 1\,000\,000$;
- алгоритмы с линейным и логарифмическим временем исполнения остаются полезными при обработке миллиарда элементов;
- в частности, алгоритм $O(\lg n)$ без труда обрабатывает любое вообразимое количество элементов.

**Таблица 2.1.** Значения времени исполнения основных функций в наносекундах. Функция lg $n$ обозначает логарифм по основанию 2 от числа $n$

| $n/f(n)$ | lg $n$ | $n$ | $n$ lg $n$ | $n^2$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|
| 10 | 0,003 мкс | 0,01 мкс | 0,033 мкс | 0,1 мкс | 1 мкс | 3,63 мс |
| 20 | 0,004 мкс | 0,02 мкс | 0,086 мкс | 0,4 мкс | 1 мс | 77,1 лет |
| 30 | 0,005 мкс | 0,03 мкс | 0,147 мкс | 0,9 мкс | 1 с | $8{,}4 \cdot 10^{15}$ лет |
| 40 | 0,005 мкс | 0,04 мкс | 0,213 мкс | 1,6 мкс | 18,3 мин | |
| 50 | 0,006 мкс | 0,05 мкс | 0,282 мкс | 2,5 мкс | 13 дней | |
| 100 | 0,007 мкс | 0,1 мкс | 0,644 мкс | 10 мкс | $4 \cdot 10^{13}$ лет | |
| 1000 | 0,010 мкс | 1,00 мкс | 9,966 мкс | 1 мс | | |
| 10 000 | 0,013 мкс | 10 мкс | 130 мкс | 100 мс | | |
| 100 000 | 0,017 мкс | 0,10 мс | 1,67 мс | 10 с | | |
| 1 000 000 | 0,020 мкс | 1 мс | 19,93 мс | 16,7 мин | | |
| 10 000 000 | 0,023 мкс | 0,01 с | 0,23 с | 1,16 дней | | |
| 100 000 000 | 0,027 мкс | 0,10 с | 2,66 с | 115,7 дней | | |
| 1 000 000 000 | 0,030 мкс | 1 с | 29,90 с | 31,7 лет | | |

Из этого можно сделать основной вывод, что, даже игнорируя постоянные множители, мы получаем превосходное общее представление о годности алгоритма для решения задачи определенного размера.

## 2.3.1. Отношения доминирования

Посредством асимптотических обозначений функции разбиваются на классы, в каждом из которых сгруппированы существенно эквивалентные функции. Например, функции $f(n) = 0{,}34n$ и $g(n) = 234{,}234n$ принадлежат к одному и тому же классу, а именно к классу порядка $\Theta(n)$. Кроме того, когда функции $f$ и $g$ принадлежат к разным классам, они являются разными относительно нашего обозначения, означая, что справедливо либо $f(n) = O(g(n))$, либо $g(n) = O(f(n))$, но не оба равенства одновременно.

Говорят, что функция с более быстрым темпом роста *доминирует* над менее быстро растущей функцией точно так же, как более быстро растущая страна в итоге начинает доминировать над отстающей. Когда функции $f$ и $g$ принадлежат к разным классам (т. е. $f(n) \neq \Theta(g(n))$), говорят, что функция $f$ *доминирует* над функцией $g$, когда $f(n) = O(g(n))$. Это отношение иногда обозначается как $g \gg f$.

К счастью, процесс базового анализа алгоритмов обычно порождает лишь небольшое количество классов функций, достаточное для покрытия почти всех алгоритмов, рассматриваемых в этой книге. Далее приводятся эти классы в порядке возрастания доминирования.

- *Функции-константы*, $f(n) = 1$.

  Такие функции могут измерять трудоемкость сложения двух чисел, распечатывания текста государственного гимна или рост таких функций, как $f(n) = \min(n, 100)$. По большому счету зависимость от параметра $n$ отсутствует.

- *Логарифмические функции*, $f(n) = \log n$.

  Логарифмическая временная сложность проявляется в таких алгоритмах, как двоичный поиск. С увеличением $n$ эти функции возрастают довольно медленно, но быстрее, чем функции-константы (которые вообще не возрастают). Логарифмы рассматриваются более подробно в *разд. 2.7*.

- *Линейные функции*, $f(n) = n$.

  Такие функции измеряют трудоемкость просмотра каждого элемента в массиве элементов один раз (или два раза, или десять раз) — например, для определения наибольшего или наименьшего элемента или для вычисления среднего значения.

- *Суперлинейные функции*, $f(n) = n \lg n$.

  Этот важный класс функций возникает в таких алгоритмах, как quicksort и mergesort. Эти функции возрастают лишь немного быстрее, чем линейные (см. табл. 2.1), но достаточно быстро, чтобы подняться к более высокому классу доминирования.

- *Квадратичные функции*, $f(n) = n^2$.

  Эти функции измеряют трудоемкость просмотра большинства или всех пар элементов в универсальном множестве из $n$ элементов. Они возникают в таких алгоритмах, как сортировка вставками или сортировка методом выбора.

- *Кубические функции*, $f(n) = n^3$.

  Эти функции перечисляют все триады элементов в универсальном множестве из $n$ элементов. Они также возникают в определенных алгоритмах динамического программирования, которые рассматриваются в *главе 10*.

- *Показательные функции, $f(n) = c^n$, константа $c > 1$.*

  Эти функции возникают при перечислении всех подмножеств множества из $n$ элементов. Как мы видели в табл. 2.1, экспоненциальные алгоритмы быстро становятся бесполезными с увеличением количества элементов $n$. Впрочем, не так быстро, как функции из следующего класса.

- *Факториальные функции, $f(n) = n!$.*

  Факториальные функции определяют все перестановки $n$ элементов.

Тонкости отношений доминирования рассматриваются в *разд. 10.2*, но в действительности вы должны помнить лишь следующее отношение:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

#### Подведение итогов

Хотя анализ алгоритмов высокого уровня может порождать экзотические функции, для большинства рассматриваемых в этой книге алгоритмов вполне достаточно лишь небольшого ассортимента функций временной сложности.

## 2.4. Работа с асимптотическими обозначениями

Для работы с асимптотическими обозначениями нужно повторить тему упрощения алгебраических выражений, которая изучается в школе. Большинство полученных в школе знаний также применимы и для асимптотических обозначений.

### 2.4.1. Сложение функций

Сумма двух функций определяется доминантной функцией, а именно:

$$f(n) + g(n) \rightarrow \Theta(\max(f(n), g(n))).$$

Это обстоятельство очень полезно при упрощении выражений. Например, из него следует, что $n^3 + n^2 + n + 1 = O(n^3)$. По сравнению с доминантным членом все остальное является несущественным.

На интуитивном уровне это объясняется таким образом. По крайней мере половина общего объема суммы $f(n) + g(n)$ должна предоставляться большей функцией. По определению при $n \rightarrow \infty$ доминантная функция будет предоставлять большую часть объема суммы функций. Соответственно, отбросив меньшую функцию, мы уменьшим значение суммы максимум в два раза, что равносильно постоянному множителю $^1/_2$. Например, если $f(n) = O(n^2)$ и $g(n) = O(n^2)$, тогда также $f(n) + g(n) = O(n^2)$.

### 2.4.2. Умножение функций

Умножение можно представлять как повторяющееся сложение. Рассмотрим умножение на любую константу $c > 0$, будь это 1,02 или 1 000 000. Умножение функции на константу не может повлиять на ее асимптотическое поведение, поскольку в анализе функции $c \cdot f(n)$ с применением обозначения «Big Oh» ($O$-большое) мы можем умно-

жить ограничивающие константы на $1/c$, чтобы привести их в соответствие. Таким образом:

$$O(cf(n)) \rightarrow O(f(n))$$
$$\Omega(cf(n)) \rightarrow \Omega(f(n))$$
$$\Theta(cf(n)) \rightarrow \Theta(f(n))$$

Конечно же, во избежание неприятностей, константа $c$ должна быть строго положительной, т. к. мы можем свести на нет даже самую быстрорастущую функцию, умножив ее на ноль.

С другой стороны, когда обе функции произведения возрастают, то обе являются важными. Функция $O(n! \log n)$ доминирует над функцией $n!$ настолько же, насколько и функция $\log n$ доминирует над константой 1. В общем,

$$O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$$
$$\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$$
$$\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$$

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Транзитивность

**ЗАДАЧА**. Доказать, что отношение $O$-большое обладает транзитивностью, т. е. если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, тогда $f(n) = O(h(n))$.

**РЕШЕНИЕ**. Работая с асимптотическими обозначениями, мы всегда обращаемся к определению. Нам нужно показать, что $f(n) \leq c_3 h(n)$ при $n > n_3$ при условии, что $f(n) \leq c_1 g(n)$ и $g(n) \leq c_2 h(n)$ при $n > n_1$ и $n > n_2$ соответственно. Из этих неравенств получаем: $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$ при $n > n_3 = \max(n_1, n_2)$. ■

# 2.5. Оценка эффективности

Как правило, общую оценку времени исполнения алгоритма можно легко получить при наличии точного описания алгоритма. В этом разделе рассматривается несколько примеров — возможно, даже более подробно, чем необходимо.

## 2.5.1. Сортировка методом выбора

Здесь мы анализируем алгоритм сортировки методом выбора. При сортировке этим методом определяется наименьший неотсортированный элемент и помещается в конец отсортированной части массива. Процедура повторяется до тех пор, пока все элементы массива не будут отсортированы. Графическая иллюстрация работы алгоритма представлена на рис. 2.4, а соответствующий код на языке C — в листинге 2.1.

**Листинг 2.1. Реализация алгоритма сортировки методом выбора на языке C**

```
void selection_sort(item_type s[], int n) {
    int i, j; /* счетчики */
    int min; /* указатель наименьшего элемента */
```

```
    for (i = 0; i < n; i++) {
        min = i;
        for (j = i + 1; j < n; j++) {
            if (s[j] < s[min]) {
                min = j;
            }
        }
        swap(&s[i], &s[min]);
    }
}
```

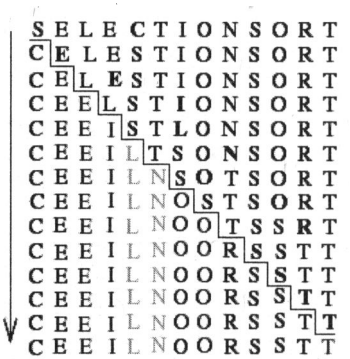



**Рис. 2.4.** Графическая иллюстрация работы алгоритма сортировки методом выбора

Внешний цикл `for` исполняется $n$ раз. Внутренний цикл исполняется $n - (i + 1)$ раз, где $i$ — счетчик внешнего цикла. Точное количество исполнений оператора `if` определяется следующей формулой:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1 .$$

Это формула сложения неотрицательных целых чисел в убывающем порядке, начиная с $n - 1$, т. е.

$$T(n) = (n - 1) + (n - 2) + (n - 3) + ... + 2 + 1.$$

Какие выводы мы можем сделать на основе такой формулы? Чтобы получить точное значение, необходимо применить методы, рассматриваемые в *разд. 2.6*. Но при работе с асимптотическими обозначениями нас интересует только *степень* выражения. Один из подходов — считать, что мы складываем $n - 1$ элементов, среднее значение которых равно приблизительно $n/2$. Таким образом мы получаем:

$$T(n) \approx (n-1)n/2 = O(n^2).$$

## Доказываем временную сложность Θ-большое

Другим подходом к анализу времени исполнения этого алгоритма будет использование верхней и нижней границ. Мы имеем не более $n$ элементов, значение каждого из которых не превышает $n - 1$. Таким образом, $T(n) \leq n(n - 1) = O(n^2)$, т. е. верхний предел определяется обозначением $O$-большое.

Нижний предел определяется обозначением Ω-большое. Возвращаясь опять к сумме, видим, что имеем $n/2$ элементов, каждый из которых больше чем $n/2$, за которыми следуют $n/2$ элементов, каждый из которых больше нуля... Соответственно:

$$T(n) \geq (n/2) \cdot (n/2) + (n/2) \cdot 0 = \Omega(n^2).$$

Вместе с результатами обозначения $O$-большое это говорит нам, что время исполнения равно $\Theta(n^2)$, т. е. сложность сортировки методом выбора является квадратичной.

По сути, преобразование анализа наихудшего случая с использованием обозначения $O$-большое в анализ с использованием обозначения Θ-большое состоит в определении экземпляра ввода, который вызывает как можно наихудшее время исполнения алгоритма. Но сортировка методом выбора выделяется среди алгоритмов сортировки тем, что она занимает абсолютно одинаковое время для всех $n!$ возможных экземпляров ввода, поскольку $T(n) = n(n - 1)/2$ для всех $n \geq 0$, $T(n) = \Theta(n2)$.

## 2.5.2. Сортировка вставками

Основное практическое правило при асимптотическом анализе гласит, что время исполнения алгоритма в наихудшем случае получается умножением наибольшего возможного количества итераций каждого вложенного цикла. Рассмотрим, например, алгоритм сортировки вставками из листинга 1.1, внутренние циклы которого приведены в листинге 2.2.

**Листинг 2.2. Внутренние циклы алгоритма сортировки вставками на языке C**

```
for (i = 1; i < n; i++) {
     j = i;
     while ((j > 0) && (s[j] < s[j - 1])) {
          swap(&s[j], &s[j - 1]);
          j = j-1;
     }
}
```

Сколько итераций осуществляет внутренний цикл `while`? На этот вопрос сложно дать однозначный ответ, поскольку цикл может быть остановлен досрочно, если произойдет выход за границы массива (`j > 0`) или элемент окажется на должном месте в отсортированной части массива (`s[j] < s[j-1]`). Так как в анализе наихудшего случая нам надо найти верхнюю границу времени исполнения, то мы игнорируем досрочное завершение и полагаем, что количество исполняемых в этом цикле итераций всегда будет $i$. Более того, мы можем упростить еще больше и допустить, что *всегда* исполнятся $n$ итераций, т. к. $i < n$. А раз внешний цикл исполняется $n$ раз, то алгоритм сортировки вставками должен быть квадратичным, т. е. $O(n^2)$.

Такой грубый анализ методом округления всегда оказывается результативным в том смысле, что полученная верхняя граница времени исполнения ($O$-большое) всегда будет правильной. Иногда этот результат может быть даже завышен в худшую сторону, т. е. в действительности время исполнения худшего случая окажется меньшим, чем результат, полученный при анализе. Тем не менее я настоятельно рекомендую этот подход в качестве основы для простого анализа алгоритмов.

### Доказываем временную сложность Θ-большое

Наихудший случай сортировки вставками происходит тогда, когда каждый вставляемый элемент нужно переместить в самое начало отсортированной области. Такая ситуация возникает, когда ввод отсортирован в порядке, обратном требуемому. Каждый из последних $n/2$ элементов ввода должен переместиться как минимум через $n/2$ элемента, чтобы занять правильное место, занимая, по крайней мере, время $(n/2)^2 = \Omega(n^2)$.

## 2.5.3. Сравнение строк

Сравнение комбинаций символов — основная операция при работе с текстовыми строками. Здесь мы рассмотрим алгоритм для реализации функции поиска определенного текста, которая является обязательной частью любого веб-браузера или текстового редактора.

**Задача.** Найти подстроку в строке.

**Вход.** Текстовая строка $t$ и строка для поиска $p$ (рис. ЦВ-2.5).

**Выход.** Содержит ли строка $t$ подстроку $p$ и, если содержит, в каком месте?

Пример практического применения этого алгоритма — поиск упоминания определенной фамилии в новостях. Для нашего экземпляра задачи текстом $t$ будет статья, а строкой $p$ для поиска — указанная фамилия.

Эта задача решается с помощью довольно простого алгоритма (листинг 2.3), который допускает, что строка $p$ может начинаться в любой возможной позиции в тексте $t$, и выполняет проверку, действительно ли, начиная с этой позиции, текст содержит искомую строку.

**Листинг 2.3. Реализация алгоритма поиска строки в тексте**

```
int findmatch(char *p, char *t) {
    int i, j; /* счетчики */
    int plen, tlen; /* длины строк */

    plen = strlen(p);
    tlen = strlen(t);

    for (i = 0; i <= (tlen-plen); i = i + 1) {
        j = 0;
        while ((j < plen) && (t[i + j] == p[j])) {
            j = j + 1;
        }
        if (j == plen) {
            return(i); /* местонахождение первого совпадения */
        }
    }
    return(-1); /* there is no match */
}
```

Каким будет время исполнения этих двух вложенных циклов в наихудшем случае? Внутренний цикл `while` исполняется максимум $m$ раз, а возможно, и намного меньше, если поиск заканчивается неудачей. Кроме оператора `while`, внешний цикл содержит еще два оператора. Внешний цикл исполняется самое большее $n - m$ раз, т. к. после продвижения направо по тексту оставшийся фрагмент будет короче искомой строки. Общая временная сложность — произведение значений оценки временной сложности внешнего и вложенного циклов, что дает нам время исполнения в худшем случае $O((n - m)(m + 2))$.

При этом мы не учитываем время, потраченное на определение длины строк с помощью функции `strlen`. Так как мы не знаем, каким образом реализована эта функция, мы можем лишь строить предположения относительно времени ее работы. Если она явно считает количество символов, пока не достигнет конца строки, то отношение между временем исполнения этой операции и длиной строки будет линейным. Таким образом, время исполнения в наихудшем случае будет равно $O(n + m + (n - m)(m + 2))$.

С помощью асимптотических обозначений это выражение можно упростить. Так как $m + 2 = \Theta(m)$, фрагмент «$+ 2$» не представляет интереса, поэтому останется только $O(n + m + (n - m)m)$. Выполнив умножение, получаем выражение $O(n + m + nm - m^2)$, которое выглядит довольно непривлекательно.

Но мы знаем, что в любой представляющей интерес задаче $n \geq m$, поскольку искомая строка $p$ не может быть подстрокой текста $t$, когда искомая строка длиннее, чем текст, в котором выполняется ее поиск. Одним из следствий этого обстоятельства является отношение $(n + m) \leq 2n = \Theta(n)$. Таким образом, формула времени исполнения для наихудшего случая упрощается дальше до $O(n + nm - m^2)$.

Еще два замечания. Обратите внимание, что $n \leq nm$, поскольку для любой представляющей интерес строки поиска $m \geq 1$. Таким образом, $n + nm = \Theta(nm)$, и мы можем опустить дополняющее $n$, упростив формулу анализа до $O(nm - m^2)$.

Кроме того, заметьте, что член «$-m^2$» отрицательный, вследствие чего он только уменьшает значение выражения внутри скобок. Так как $O$-большое задает верхнюю границу, то любой отрицательный член можно удалить, не искажая оценку верхней границы. Неравенство $n \geq m$ подразумевает, что $mn \geq m^2$, поэтому отрицательный член недостаточно большой, чтобы аннулировать оставшийся член. Таким образом, время исполнения этого алгоритма в худшем случае можно выразить просто как $O(nm)$.

Накопив достаточно опыта, вы сможете выполнять такой анализ алгоритмов в уме, даже не прибегая к изложению алгоритма в письменной форме. В конце концов, одной из составляющих разработки алгоритма для решения предоставленной задачи является перебор в уме разных способов и выбор самого лучшего из них. Такое умение приобретается с опытом, но если у вас недостаточно практики и вы не понимаете, почему время исполнения этого алгоритма равно $O(f(n))$, то сначала распишите подробно алгоритм, а потом выполните последовательность логических рассуждений, наподобие продемонстрированных в этом разделе.

## Доказываем время исполнения по Θ-большое

Предоставленный анализ выдает квадратичное время верхнего предела времени исполнения этого простого алгоритма сопоставления с образцом. Чтобы доказать обозна-

чение Θ-большое, нужно привести пример такого алгоритма, время исполнения которого действительно занимает время $\Omega(mn)$. Рассмотрим, что происходит при поиске образца подстроки $p = aaaa \ldots aaab$, состоящей из $m - 1$ букв $a$ и буквы $b$ в конце, в тексте $t = aaaa \ldots aaaa$, состоящем из $n$ *букв* $a$.

При каждом размещении подстроки образца на строке текста цикл `while` успешно сопоставит первые $m - 1$ символов перед тем, как потерпеть неудачу с последним символом. Подстроку образца $p$ можно разместить в строке поиска $t$ в $n - m + 1$ возможных позициях, не выходя за конец этой строки, поэтому время исполнения будет

$$(n - m + 1)(m) = mn - m2 + m = \Omega(mn).$$

Таким образом, этот алгоритм поиска строки имеет наихудший случай времени исполнения $\Theta(nm)$. Тем не менее существуют и более быстрые алгоритмы. В *разд. 6.7* мы рассмотрим один из таких алгоритмов с линейным ожидаемым временем исполнения.

## 2.5.4. Умножение матриц

При анализе алгоритмов с вложенными циклами часто приходится иметь дело с вложенными операциями суммирования. Рассмотрим задачу умножения матриц:

**Задача.** Умножить матрицы.

**Вход.** Матрица $A$ (размером $x \times y$) и матрица $B$ (размером $y \times z$).

**Выход.** Матрица $C$ размером $x \times z$, где $C[i][j]$ является скалярным произведением строки $i$ матрицы $A$ и столбца $j$ матрицы $B$.

Умножение матриц является одной из основных операций в линейной алгебре (пример задачи на умножение матриц рассматривается в *разд. 16.3*). А в листинге 2.4 приводится пример реализации простого алгоритма для умножения матриц с использованием трех вложенных циклов.

**Листинг 2.4. Умножение матриц**

```
for (i = 1; i <= a->rows; i++) {
     for (j = 1; j <= b->columns; j++) {
          c->m[i][j] = 0;
          for (k = 1; k <= b->rows; k++) {
               c->m[i][j] += a->m[i][k] * b->m[k][j];
          }
     }
}
```

Каким образом нам подойти к анализу временной сложности этого алгоритма? К этому времени вы должны распознать, что три вложенных цикла означают $O(n^3)$, но давайте будем точными. Количество операций умножения $M(x, y, z)$ определяется такой формулой:

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1 .$$

Суммирование выполняется справа налево. Сумма $z$ единиц равна $z$, поэтому можно написать:

$$M(x,y,z)=\sum_{i=1}^{x}\sum_{j=1}^{y}z\,.$$

Сумма $y$ членов $z$ вычисляется так же просто. Она равна $yz$, тогда $M(x,y,z)=\sum_{i=1}^{x}yz$.

Наконец, сумма $x$ членов $yz$ равна $xyz$.

Таким образом, время исполнения этого алгоритма для умножения матриц равно $O(xyz)$. В общем случае, когда все три измерения матриц одинаковы, это сводится к $O(n^3)$. Этот же анализ применим и к нижней границе $\Omega(n^3)$, т. к. количество итераций циклов зависит от размерности матриц. Простое умножение матриц оказывается кубическим алгоритмом со временем исполнения, равным $\Theta(n^3)$. При всем этом существуют и более быстрые алгоритмы, несколько примеров которых приводятся в *разд. 16.3*.

## 2.6. Суммирование

Математические формулы суммирования важны по двум причинам. Во-первых, к ним часто приходится прибегать в процессе анализа алгоритмов. А во-вторых, процесс доказательства правильности формулы суммирования представляет собой классический пример применения математической индукции. В конце этой главы дается несколько упражнений, в которых требуется доказать формулу с помощью индукции. Чтобы сделать эти упражнения более понятными, напомню основные принципы суммирования.

Формулы суммирования представляют собой краткие выражения, описывающие сложение сколь угодно большой последовательности чисел. В качестве примера можно привести такую формулу:

$$\sum_{i=1}^{n}f(i)=f(1)+f(2)+\ldots+f(n)\,.$$

Суммирование многих алгебраических функций можно выразить простыми формулами в замкнутой форме. Например, поскольку сумма $n$ единиц равна $n$, то

$$\sum_{i=1}^{n}1=n\,.$$

Для четного $n$ сумму первых $n=2k$ целых чисел можно выразить, объединив парами $i$-е и $(n-i+1)$-е целые числа следующим образом:

$$\sum_{i=1}^{n}i=\sum_{i=1}^{k}\bigl(i+(2k-i+1)\bigr)=k(2k+1)=n(n+1)/2.$$

Применив немного более кропотливый анализ, такой же результат мы получим и для нечетных чисел.

Очень пригодится в области анализа алгоритмов умение распознавать два основных класса формул суммирования:

- *сумма степени целых чисел.*

  Арифметическую прогрессию в виде формулы

$$S(n) = \sum_{i=1}^{n} i = n(n+1)/2$$

  можно встретить в анализе алгоритма сортировки методом выбора. По большому счету важным фактом является наличие квадратичной суммы, а не то, что константа равняется $^1/_2$. В общем, для $p \geq 0$:

$$S(n, p) = \sum_{i}^{n} i^p = \Theta(n^{p+1}).$$

  Таким образом, сумма квадратов кубичная, а сумма кубов — «четверичная» (если вы не против употребления такого слова).

  Для $p < -1$ сумма $S(n, p)$ всегда стремится к константе, когда $n \to \infty$, а для $p \geq 0$ она расходится. Между этими двумя случаями представляет интерес случай гармонических чисел:

$$H(n) = \sum_{i=1}^{n} 1/i = \Theta(\log n).$$

- *сумма геометрической прогрессии.*

  В геометрических прогрессиях индекс цикла играет роль показателя степени, т. е.

$$G(n, a) = \sum_{i=0}^{n} a^i = a(a^{n+1} - 1)/(a - 1).$$

  Сумма прогрессии зависит от ее знаменателя, т. е. от числа $a$. При $|a| < 1$ сумма $G(n, a)$ стремится к константе, когда $n \to \infty$.

  Такая сходимость последовательности оказывается большим подспорьем в анализе алгоритмов. Это означает, что если количество элементов растет линейно, то их сумма необязательно будет расти линейно, а может быть ограничена константой. Например, $1 + 1/2 + 1/4 + 1/8 + \ldots \leq 2$ независимо от количества элементов последовательности.

  При $a > 1$ сумма стремительно возрастает при добавлении каждого нового элемента — например: $1 + 2 + 4 + 8 + 16 + 32 = 63$.

  В самом деле, для $a > 1$ $G(n, a) = \Theta(a^{n+1})$.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Формулы факториала

**ЗАДАЧА.** Докажите методом индукции, что $\sum_{i=1}^{n} i \times i! = (n+1)! - 1$.

**РЕШЕНИЕ.** Индукционная парадигма прямолинейна: сначала подтверждаем базовый случай. При $n = 0$ сумма будет пустая, что согласно определению равно 0. Альтернативно, можно использовать $n = 1$:

$$\sum_{i=1}^{1} i \times i! = 1 = (1+1)! - 1 = 2 - 1 = 1.$$

Теперь допускаем, что это утверждение верно для всех чисел вплоть до $n$. Для доказательства общего случая $n + 1$ видим, что если мы вынесем наибольший член из-под знака суммы:

$$\sum_{i=1}^{n+1} i \times i! = (n+1) \times (n+1)! + \sum_{i=1}^{n} i \times i!,$$

то получим левую часть нашего индуктивного допущения. Заменяя правую часть, получаем:

$$\begin{aligned}\sum_{i=1}^{n+1} i \times i! &= (n+1) \times (n+1)! + (n+1)! - 1 = \\ &= (n+1)! \times ((n+1)+1) - 1 = \\ &= (n+2)! - 1.\end{aligned}$$

Этот общий прием выделения наибольшего члена суммы для выявления экземпляра индуктивного допущения лежит в основе всех таких доказательств. ■

# 2.7. Логарифмы и их применение

Слово «логарифм» — почти анаграмма слова «алгоритм». Но наш интерес к логарифмам вызван не этим обстоятельством. Возможно, что кнопка калькулятора с обозначением «log» — единственное место, где вы сталкиваетесь с логарифмами в повседневной жизни. Также возможно, что вы уже не помните назначение этой кнопки. *Логарифм* — это функция, обратная показательной. То есть выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Более того, из этой равнозначности следует, что $b^{\log_b y} = y$.

Показательные функции возрастают чрезвычайно быстро, что может засвидетельствовать любой, кто когда-либо выплачивал долг по кредиту. Соответственно, функции, обратные показательным, т. е. логарифмы, возрастают довольно медленно. Логарифмические функции возникают в любом процессе, содержащем деление пополам. Давайте рассмотрим несколько таких примеров.

## 2.7.1. Логарифмы и двоичный поиск

Двоичный поиск является хорошим примером алгоритма с временной логарифмической сложностью $O(\log n)$. Чтобы найти определенного человека по имени $p$ в телефонной книге[2], содержащей $n$ имен, мы сравниваем имя $p$ с выбранным именем посередине книги (т. е. с $n/2$-м именем) — скажем, Monroe, Marilyn. Независимо от того, находится ли имя $p$ перед выбранным именем (например, Dean, James) или после него (например, Presley, Elvis), после этого сравнения мы можем отбросить половину всех имен в книге. Этот процесс повторяется с половиной книги, содержащей искомое имя, и далее, пока не останется всего лишь одно имя, которое и будет искомым. По определению количество таких делений равно $\log_2 n$. Таким образом, чтобы найти любое имя в телефонной книге Манхэттена (содержащей миллион имен), достаточно выполнить всего лишь двадцать сравнений. Потрясающе, не так ли?

---

[2] Если вы не знаете, что такое телефонная книга, спросите у своих бабушки или дедушки.

Идея двоичного поиска является одной из наиболее плодотворных в области разработки алгоритмов. Его мощь становится очевидной, если мы представим, что нам нужно найти определенного человека в неотсортированном телефонном справочнике.

## 2.7.2. Логарифмы и деревья

Двоичное дерево высотой в один уровень может иметь две концевые вершины (листа), а дерево высотой в два уровня может иметь до четырех листов. Какова высота $h$ двоичного дерева, имеющего $n$ листов? Обратите внимание, что количество листов удваивается при каждом увеличении высоты дерева на один уровень. Таким образом, зависимость количества листов $n$ от высоты дерева $h$ выражается формулой $n = 2^h$, откуда следует, что $h = \log_2 n$.

Теперь перейдем к общему случаю. Рассмотрим деревья, которые имеют $d$ потомков (для двоичных деревьев $d = 2$). Такое дерево высотой в один уровень может иметь $d$ количество листов, а дерево высотой в два уровня может иметь $d^2$ количество листов. Количество листов на каждом новом уровне можно получить, умножая на $d$ количество листов предыдущего уровня. Таким образом, количество листов $n$ выражается формулой $n = d^h$, т. е. высота находится по формуле $h = \log_d n$ (рис. 2.6).

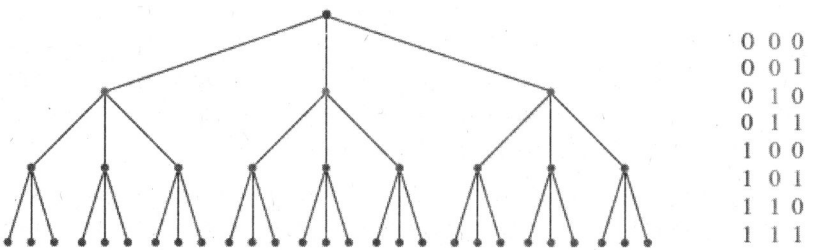



**Рис. 2.6.** Дерево высотой $h$ и с количеством потомков $d$ для каждого узла имеет $d^h$ листьев. В нашем случае $h = 3$, $d = 3$ (*слева*). Количество комбинаций битов (*справа*) растет экспоненциально с увеличением длины комбинации. Эти комбинации можно описать как пути от корня к листу в *двоичном* дереве высотой $h = 3$

Из сказанного можно сделать вывод, что деревья небольшой высоты могут иметь очень много листьев. Это обстоятельство является причиной того, что двоичные деревья лежат в основе всех быстро обрабатываемых структур данных.

## 2.7.3. Логарифмы и биты

Предположим, имеются две однобитовые комбинации (0 и 1), четыре двухбитовые комбинации (00, 01, 10 и 11) и восемь трехбитовых комбинаций (см. рис. 2.6, *справа*). Сколько битов $w$ потребуется, чтобы представить любую из $n$ возможных разных комбинаций, будь то один из $n$ элементов или одно из целых чисел от 0 до $n - 1$?

Ключевым наблюдением здесь является то обстоятельство, что нужно иметь, по крайней мере, $n$ разных битовых комбинаций длиной $w$. Так как количество разных битовых комбинаций удваивается с добавлением каждого бита, то нам нужно по крайней мере $w$ битов, где $2^w = n$, т. е. нам нужно $w = \log_2 n$ битов.

## 2.7.4. Логарифмы и умножение

Логарифмы имели особенно большую важность до распространения карманных калькуляторов. Применение логарифмов было самым легким способом умножения больших чисел вручную: либо с помощью логарифмической линейки, либо с использованием таблиц.

Но и сегодня логарифмы остаются полезными для выполнения операций умножения, особенно для возведения в степень. Вспомните, что $\log_a(xy) = \log_a(x) + \log_a(y)$, т. е. логарифм произведения равен сумме логарифмов сомножителей. Прямым следствием этого является формула

$$\log_a n^b = b \cdot \log_a n .$$

Выясним, как вычислить $a^b$ для любых $a$ и $b$, используя функции $\exp(x)$ и $\ln(x)$ на карманном калькуляторе, зная, что $\exp(x) = e^x$ и $\ln(x) = \log_e(x)$. Таким образом,

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a) .$$

То есть задача сводится к одной операции умножения с однократным вызовом каждой из этих функций.

## 2.7.5. Быстрое возведение в степень

Допустим, что нам нужно вычислить *точное* значение $a^n$ для достаточно большого значения $n$. Такие задачи в основном возникают в криптографии при проверке числа на простоту (см. *разд. 16.8*). Проблемы с точностью не позволяют нам воспользоваться ранее рассмотренной формулой возведения в степень.

Самый простой алгоритм выполняет $n - 1$ операций умножения ($a \times a \times \ldots \times a$). Но можно указать лучший способ решения этой задачи, приняв во внимание, что $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. Если $n$ четное, тогда $a^n = (a^{n/2})^2$. А если $n$ нечетное, то тогда $a^n = a(a^{\lfloor n/2 \rfloor})^2$. В любом случае значение показателя степени было уменьшено наполовину, а вычисление сведено самое большее к двум операциям умножения. Таким образом, для вычисления конечного значения будет достаточно $O(\lg n)$ операций умножения. Псевдокод соответствующего алгоритма приведен в листинге 2.5.

**Листинг 2.5. Алгоритм быстрого возведения в степень**

```
function power(a, n)
    if (n = 0) return(1)
    x = power (a, ⌊n/2⌋)
    if (n is even) then return(x²)
        else return(a × x²)
```

Этот простой алгоритм иллюстрирует важный принцип: *разделяй* и *властвуй*. Разделение задачи на (по возможности) равные подзадачи всегда окупается. Когда значение $n$ отлично от 2, то входные данные не всегда можно разделить точно пополам, но разница в один элемент между двумя половинами, как показано здесь, не вызовет серьезного нарушения баланса.

## 2.7.6. Логарифмы и сложение

Гармоническое число представляет собой особый случай суммы степени целых чисел, а именно $H(n) = S(n, -1)$. Это сумма обратных величин первых $n$ последовательных чисел натурального ряда:

$$H(n) = \sum 1/i = \Theta(\log n).$$

Гармонические числа помогают объяснить, откуда берутся логарифмы в алгебраических операциях. Например, ключевым фактором в анализе среднего случая сложности алгоритма быстрой сортировки quicksort является следующее суммирование:

$$S(n) = n\sum_{i=1}^{n} 1/i\ .$$

Применение тождества гармонического числа сразу же упрощает это выражение до $\Theta(n \log n)$.

## 2.7.7. Логарифмы и система уголовного судопроизводства

В табл. 2.2 приводится пример использования логарифмов.

**Таблица 2.2.** Рекомендуемые наказания в федеральных судах США за преступления финансового мошенничества

| Понесенные убытки | Повышение уровня наказания |
|---|---|
| (A) 2000 долларов или меньше | Уровень не повышается |
| (B) Свыше 2000 долларов | Повысить на один уровень |
| (C) Свыше 5000 долларов | Повысить на два уровня |
| (D) Свыше 10 000 долларов | Повысить на три уровня |
| (E) Свыше 20 000 долларов | Повысить на четыре уровня |
| (F) Свыше 40 000 долларов | Повысить на пять уровней |
| (G) Свыше 70 000 долларов | Повысить на шесть уровней |
| (H) Свыше 120 000 долларов | Повысить на семь уровней |
| (I) Свыше 200 000 долларов | Повысить на восемь уровней |
| (J) Свыше 350 000 долларов | Повысить на девять уровней |
| (K) Свыше 500 000 долларов | Повысить на десять уровней |
| (L) Свыше 800 000 долларов | Повысить на одиннадцать уровней |
| (M) Свыше 1 500 000 долларов | Повысить на двенадцать уровней |
| (N) Свыше 2 500 000 долларов | Повысить на тринадцать уровней |
| (O) Свыше 5 000 000 долларов | Повысить на четырнадцать уровней |

**Таблица 2.2** *(окончание)*

| Понесенные убытки | Повышение уровня наказания |
|---|---|
| (P) Свыше 10 000 000 долларов | Повысить на пятнадцать уровней |
| (Q) Свыше 20 000 000 долларов | Повысить на шестнадцать уровней |
| (R) Свыше 40 000 000 долларов | Повысить на семнадцать уровней |
| (S) Свыше 5 000 000 долларов | Повысить на восемнадцать уровней |

Эта таблица из Федерального руководства по вынесению наказаний используется во всех федеральных судах Соединенных Штатов. Изложенные в ней рекомендуемые уровни наказания являются попыткой стандартизировать выносимые приговоры, чтобы за преступления одинаковой категории разные суды приговаривали к одинаковому наказанию. Для этого специальная комиссия выработала сложную функцию для оценки тяжести преступления и соотнесения его со сроком заключения. Эта функция реализована в табл. 2.2, где приведены соотношения между понесенным убытком в долларах и соответствующим повышением базового наказания. Обратите внимание, что наказание повышается на один уровень при приблизительном удвоении суммы незаконно присвоенных денег. Это означает, что уровень наказания (который практически линейно связан со сроком заключения) возрастает логарифмически по отношению к сумме незаконно присвоенных денег.

Задумайтесь на минуту о последствиях этого обстоятельства. Несомненно, многие нечистые на руку руководители фирм тоже задумывались на этот счет. Все сказанное означает, что общий срок заключения возрастает *чрезвычайно* медленно по отношению к росту суммы украденных денег. Срок заключения за пять ограблений винно-водочных магазинов на общую сумму 50 000 долларов будет значительно выше, чем за однократное завладение посредством мошеннических операций суммой в 1 000 000 долларов. Соответственно, выгода от обогащения подобным образом на действительно крупные суммы будет еще больше. Мораль логарифмического роста функций ясна: уж если воровать, так миллионы[3].

#### *Подведение итогов*

Логарифмические функции возникают при решении задач с повторяющимся делением или удваиванием входных данных.

## 2.8. Свойства логарифмов

Как мы уже видели, выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Член $b$ называется *основанием* логарифма. Особый интерес представляют следующие основания логарифмов:

[3] «Жизнь имитирует искусство». После публикации этого примера в предыдущем издании книги со мной связался представитель Комиссии по определению приговоров США с просьбой помочь с улучшением этого руководства.

- *основание $b = 2$.*

  *Двоичный логарифм*, обычно обозначаемый как $\lg x$, является логарифмом по основанию 2. Мы уже видели, что логарифмами с этим основанием выражается временная сложность алгоритмов, использующих многократное деление пополам (т. е. двоичный поиск) или умножение на два (т. е. листья деревьев). В большинстве случаев, когда речь идет о применении логарифмов в алгоритмах, подразумеваются двоичные логарифмы;

- *основание $b = e$.*

  *Натуральный логарифм*, обычно обозначаемый как $\ln x$, является логарифмом по основанию $e = 2.71828...$ Обратной к функции натурального логарифма является экспоненциальная функция $\exp(x) = e^x$. Суперпозиция этих функций дает нам функцию тождественности:

$$\exp(\ln x) = x \text{ и } \ln(\exp x) = x;$$

- *основание $b = 10$.*

  Менее распространенными на сегодняшний день являются логарифмы по основанию 10, или *десятичные логарифмы*. До появления карманных калькуляторов логарифмы с этим основанием использовались в логарифмических линейках и таблицах алгоритмов.

Мы уже видели одно важное свойство логарифмов, а именно, что

$$\log_a(xy) = \log_a(x) + \log_a(y).$$

Другой важный факт, который нужно запомнить: логарифм по одному основанию легко преобразовать в логарифм по другому основанию. Для этого служит следующая формула:

$$\log_a b = \frac{\log_c b}{\log_c a}.$$

Таким образом, чтобы изменить основание $a$ логарифма $\log_a b$ на $c$, логарифм просто нужно умножить на $\log_c a$. В частности, функцию натурального логарифма можно с легкостью преобразовать в функцию десятичного логарифма и наоборот.

Из этих свойств логарифмов следуют два важных с арифметической точки зрения следствия.

- *Основание логарифма не оказывает значительного влияния на скорость роста функции.* Сравните следующие три значения: $\log_2(1\,000\,000) = 19{,}9316$, $\log_3(1\,000\,000) = 12{,}5754$ и $\log_{100}(1\,000\,000) = 3$. Как видите, большое изменение в основании логарифма сопровождается малыми изменениями в значении логарифма. Чтобы изменить у логарифма основание $a$ на $c$, первоначальный логарифм нужно умножить на $\log_c a$. Этот коэффициент поглощается нотацией «$O$-большое», когда $a$ и $c$ являются константами. Таким образом, игнорирование основания логарифма при анализе алгоритма обычно оправдано.

- *Логарифмы уменьшают значение любой функции.* Скорость роста логарифма любой полиномиальной функции определяется как $O(\lg n)$. Это вытекает из равенства

$$\log_a n^b = b \cdot \log_a n.$$

Эффективность двоичного поиска в широком диапазоне задач является прямым следствием этого свойства. Обратите внимание, что двоичный поиск в отсортированном массиве из $n^2$ элементов требует всего лишь вдвое больше сравнений, чем в массиве из $n$ элементов.

Логарифмы эффективно уменьшают значение любой функции. С факториалами трудно выполнять какие-либо вычисления, если не пользоваться логарифмами, и тогда формула

$$n! = \prod_{i=1}^{n} i \rightarrow \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

предоставляет еще одно основание для появления логарифмов в анализе алгоритмов.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Важно ли деление точно пополам

**ЗАДАЧА**. Насколько больше операций сравнения потребуется при двоичном поиске, чтобы найти совпадение в списке из миллиона элементов, если вместо деления точно пополам делить список в отношении 1/3 к 2/3?

**РЕШЕНИЕ.** Не намного больше, чем при делении списка точно пополам, а именно $\log_{3/2}(1\,000\,000) \approx 35$ операций сравнения в самом худшем случае, что не намного больше, чем $\log_2(1\,000\,000) \approx 20$ операций сравнения при делении пополам. Изменение основания логарифма не влияет на асимптотическую сложность. Эффективность алгоритма двоичного поиска определяется логарифмической временной сложностью, а не основанием логарифма. ■

## 2.9. История из жизни. Загадка пирамид

По выражению его глаз я должен был догадаться, что услышу что-то необычное.

— Мы хотим выполнять вычисления до 1 000 000 000 на многопроцессорном суперкомпьютере, но для этого нам нужен более быстрый алгоритм.

Мне приходилось видеть такой взгляд раньше. Глаза собеседника помутнели от того, что этот человек уже давно ни о чем не думал, полагая, что мощные суперкомпьютеры избавляют его от необходимости разрабатывать сложные алгоритмы. Во всяком случае это было так, пока задача не стала достаточно сложной.

— Я работаю с лауреатом Нобелевской премии над решением знаменитой задачи теории чисел с помощью компьютера. Вы знакомы с проблемой Уоринга?

Я имел некоторое представление о теории чисел.

— Конечно. В проблеме Уоринга ставится вопрос, можно ли выразить каждое целое число, по крайней мере одним способом, как сумму квадратов самое большее четырех целых чисел. Например, $78 = 8^2 + 3^2 + 2^2 + 1^2 = 7^2 + 5^2 + 2^2$. Я помню, как в институтском курсе теории чисел мне приходилось доказывать, что для выражения любого целого числа будет достаточно квадратов четырех целых чисел. Да, это знаменитая задача, но она была решена около 200 лет тому назад.

— Нет, нас интересует другая версия проблемы Уоринга, с использованием пирамидальных чисел. *Пирамидальным* называется число, которое можно представить в виде $(m^3 - m)/6$ при $m \geq 2$. Примером нескольких первых пирамидальных чисел будут 1, 4, 10, 20, 35, 56, 84, 120, 165. С 1928 года существует гипотеза, что любое целое число можно выразить суммой самое большее пяти пирамидальных чисел. Мы хотим с помощью суперкомпьютера доказать эту гипотезу для всех чисел от 1 до 1 000 000 000.

— Миллиард любых вычислений займет значительное время, — предупредил я. — Критичным будет время, затраченное на вычисление минимального представления каждого числа, т. к. это придется делать миллиард раз. Вы думали, какой тип алгоритма использовать?

— Мы уже написали свою программу и испытали ее на многопроцессорном суперкомпьютере. На небольших числах она работает очень быстро, но при работе с числами, большими 100 000, время значительно увеличивается.

Все ясно, — подумал я. Этот «компьютероман» открыл асимптотический рост. Никаких сомнений, что он использовал алгоритм квадратичной временной сложности и столкнулся с проблемами, как только число $n$ стало достаточно большим.

— Нам нужна более быстрая программа, чтобы дойти до одного миллиарда. Можете ли вы помочь нам с этой задачей? Конечно же, мы будем выполнять ее на нашем многопроцессорном суперкомпьютере.

Поскольку я люблю такой тип задач — разработку алгоритмов для ускорения времени работы программ, — я согласился подумать над этим и принялся за работу.

Я начал с просмотра программы, написанной моим посетителем. Он создал массив всех $\Theta(n^{1/3})$ пирамидальных чисел от 1 до $n$ включительно[4]. Для каждого числа $k$ в этом диапазоне методом полного перебора выполнялась проверка, являлось ли оно суммой двух пирамидальных чисел. При отрицательном результате проверки выполнялась проверка, было ли число суммой трех пирамидальных чисел, потом четырех и, наконец, пяти, пока не находился ответ. Приблизительно 45% целых чисел можно выразить как сумму трех пирамидальных чисел. Большинство из оставшихся 55% чисел можно представить в виде суммы четырех пирамидальных чисел и, как правило, разными способами. Известно только 241 целое число, представляемое суммой пяти пирамидальных чисел, самое большое из которых равно 343 867. Для примерно половины из $n$ чисел этот алгоритм проверял все трехэлементные комбинации и, по крайней мере, некоторые четырехэлементные. Таким образом, общее время исполнения этого алгоритма было как минимум $O(n \times (n^{1/3})^3) = O(n^2)$, где $n = 1\,000\,000\,000$. Неудивительно, что на больших числах (превышающих 100 000) эта программа замедляла работу.

Любое решение, работающее на больших числах значительно быстрее предложенного, должно избегать явной проверки всех трехэлементных комбинаций. Для каждого значения $k$ нам нужно наименьшее множество пирамидальных чисел, сумма которых в точности равна $k$. Эта задача называется *задачей о рюкзаке* и рассматривается

---

[4] Почему $n^{1/3}$? Вспомните, что пирамидальные числа выражаются формулой $(m^3 - m)/6$. Наибольшее число $m$, при котором результат не превышает $n$, приблизительно равно $\sqrt[3]{6n}$, поэтому количество таких чисел выражается формулой $\Theta(n^{1/3})$.

в *разд. 16.10*. В нашем случае весу предметов соответствует набор пирамидальных чисел, не превышающих $n$, с тем дополнительным ограничением, что рюкзак вмещает ровно $k$ предметов (т. е. чисел, в нашем случае).

В стандартном подходе к решению задачи о рюкзаке заранее вычисляются суммы меньших подмножеств, которые потом используются в вычислении больших подмножеств. Если у нас имеется таблица, содержащая суммы двух чисел, и мы хотим узнать, можно ли выразить число $k$ в виде суммы трех чисел, то можем перефразировать постановку задачи и спросить, можно ли выразить число $k$ в виде суммы одного числа и одной из сумм в нашей таблице.

Поэтому мне нужно было составить таблицу всех целых чисел меньших чем $n$, которые можно выразить в виде суммы двух из 1816 нетривиальных пирамидальных чисел, меньших чем 1 000 000 000. Таких чисел может быть самое большее $1816^2 = 3\,297\,856$. Более того, если мы уберем повторяющиеся суммы и суммы, большие чем целевое число, у нас останется меньше чем половина чисел. Создание отсортированного массива этих чисел не составит никакого труда. Назовем эту отсортированную структуру данных *таблицей сумм двух пирамидальных чисел* (*two*-table).

Поиск минимального разложения заданного числа $k$ начинается с проверки, является ли оно одним из 1816 пирамидальных чисел. Если не является, то тогда выполняется проверка, не находится ли оно в таблице сумм двух пирамидальных чисел. Чтобы проверить, можно ли выразить число $k$ в виде суммы трех пирамидальных чисел, нужно было всего лишь проверить, что $k - p[i]$ находится в таблице сумм двух пирамидальных чисел при $1 \le i \le 1816$. Эту проверку можно было быстро выполнить посредством двоичного поиска. Чтобы проверить, можно ли выразить число $k$ в виде суммы четырех пирамидальных чисел, нужно было всего лишь проверить, что $k - two[i]$ находится в таблице сумм двух пирамидальных чисел для любого $1 \le i \le |two|$. Но т. к. почти любое число $k$ можно выразить как сумму многих комбинаций четырех пирамидальных чисел, эта проверка не займет много времени, и доминирующей составляющей времени общей проверки будет время, затраченное на проверку трехэлементных сумм. Временная сложность проверки, является ли число $k$ суммой трех пирамидальных чисел, оценивается как $O(n^{1/3}\lg n)$, а для всего множества $n$ целых чисел — как $O(n^{4/3}\lg n)$. По сравнению с временной сложностью $O(n^2)$ алгоритма клиента для $n = 1\,000\,000\,000$, мой алгоритм был в 30 000 раз быстрее.

Первый прогон реализации этого алгоритма на моем далеко не новом компьютере Sparc ELC занял около 20 минут для $n = 1\,000\,000$. После этого я экспериментировал с представлением наборов чисел с разными структурами данных и с разными алгоритмами для поиска в этих структурах. В частности, я попробовал использовать вместо отсортированных массивов хеш-таблицы и двоичные векторы, поэкспериментировал и с разновидностями двоичного поиска, такими как интерполяционный поиск (см. *разд. 17.2*). Наградой за эту работу стало менее чем трехминутное время исполнения программы на множестве чисел $n = 1\,000\,000$, что в шесть раз лучше времени исполнения первоначальной программы.

Завершив основную работу, я занялся настройкой программы, чтобы немного повысить ее производительность. В частности, поскольку 1 — это пирамидальное число, то для любого числа $k$, когда $k - 1$ было суммой трех пирамидальных чисел, я не вычислял

сумму четырех пирамидальных чисел. В результате использования только этого приема общее время исполнения программы было сокращено еще на 10%. Наконец, с помощью профилировщика я выполнил несколько низкоуровневых настроек, чтобы еще чуть-чуть повысить производительность. Например, заменив лишь одну вызываемую процедуру встроенным кодом, я сбросил еще 10% с времени исполнения.

После этого я передал программу заказчику. Он использовал ее самым неподходящим образом, о чем я расскажу в *разд. 5.8*.

Подготавливая к публикации эту историю из жизни, я вытащил из архивов и проверил на исполнение ту программу, которой сейчас больше лет, чем моим текущим аспирантам. Даже в однопоточном режиме она исполнилась за 1,113 секунды. Включив оптимизатор компилятора, удалось сократить этот показатель до всего лишь 0,334 секунды. Вот поэтому нужно не забывать включать оптимизатор компилятора при поиске способов ускорения исполнения программы. Время исполнения этого кода ускорилось в сотни раз, хотя и не предпринималось ничего, кроме ожидания в течение 25 лет улучшений аппаратного обеспечения. Собственно говоря, сервер в нашей лаборатории сейчас может проверить миллиард чисел чуть меньше, чем за три часа (174 минуты и 28,4 секунды), работая лишь в однопоточном режиме. Более того, этот код исполняется за 9 часов, 37 минут и 34,8 секунды на убогом ноутбуке MacBook Apple, клавиши которого выпадают, когда я набираю на нем материал этой книги.

Основная цель моего рассказа состоит в том, чтобы показать громадный потенциал повышения скорости вычислений за счет улучшения эффективности алгоритмов, по сравнению с весьма скромным повышением производительности, получаемым за счет установки более дорогого оборудования. Применив более эффективный алгоритм, я повысил скорость вычислений приблизительно в 30 тысяч раз. Суперкомпьютер моего заказчика, стоивший миллион долларов, был оснащен (в то время) 16 процессорами, каждый из которых мог выполнять целочисленные вычисления в пять раз быстрее, чем мой настольный компьютер стоимостью в 3000 долларов. Но при использовании всей этой техники скорость выполнения моей программы возросла менее чем в 100 раз. Очевидно, что в рассматриваемом случае применение более эффективного алгоритма имеет преимущество над использованием более мощного оборудования, что справедливо для любой задачи с достаточно большим вводом.

## 2.10. Анализ высшего уровня (*)

В идеальном случае мы все умели бы свободно обращаться с математическими методами асимптотического анализа. Точно так же в идеале мы все были бы богатыми и красивыми. А поскольку жизнь далека от идеала, математические методы асимптотического анализа требуют определенных знаний и практики.

В этом разделе выполняется обзор основных методов и функций, применяемых в анализе алгоритмов на высшем уровне. Это факультативный материал — он не используется нигде в *первой* части книги. В то же самое время знание этого материала немного поспособствует пониманию некоторых функций временной сложности, рассматриваемых во *второй* ее части.

## 2.10.1. Малораспространенные функции

Основные классы функций временной сложности алгоритмов были представлены в *разд. 2.3.1*. Но в расширенном анализе алгоритмов также возникают и менее распространенные функции. И хотя такие функции нечасто встречаются в этой книге, вам будет полезно знать, что они означают и откуда происходят. Далее приводятся некоторые из таких функций и их краткое описание.

- *Обратная функция Аккермана:* $f(n) = \alpha(n)$.

  Эта функция появляется в подробном анализе нескольких алгоритмов, в особенности в подробном анализе структуры данных объединение — поиск, рассматриваемой в *разд. 8.1.3*. Будет достаточно воспринимать эту функцию как технический термин для самой медленно возрастающей функции сложности алгоритмов. В отличие от функции-константы, $f(n) = 1$, $\alpha(n)$ достигает бесконечности при $n \to \infty$, но она определенно не торопится сделать это. Для любого значения $n$ в физической вселенной значение $\alpha(n)$ будет меньше 5, т. е. $\alpha(n) < 5$.

- $f(n) = \log\log n$.

  Смысл функции «log log» очевиден по ее имени — это логарифм логарифма числа $n$. Одним из естественных примеров ее возникновения будет двоичный поиск в отсортированном массиве из всего лишь $\lg n$ элементов.

- $f(n) = \log n / \log\log n$.

  Эта функция возрастает немного медленнее, чем функция $\log n$, т. к. она содержит в знаменателе еще более медленно растущую функцию.

  Чтобы понять, как возникла эта функция, рассмотрим корневое дерево с количеством потомков $d$, имеющее $n$ листьев. Высота двоичных корневых деревьев, т. е. деревьев, у которых $d = 2$, определяется следующей формулой:

  $$n = 2^h \to h = \lg n,$$

  получающейся в результате логарифмирования обеих частей равенства. Рассмотрим теперь высоту дерева, когда количество потомков равно $d = \log n$. Тогда высота определяется такой формулой: $n = (\log n) \to h = \log n / \log\log n$.

- $f(n) = \log^2 n$.

  Это произведение двух логарифмических функций, т. е. $(\log n) \times (\log n)$. Такая функция может возникнуть, если мы хотим подсчитать просмотренные биты в процессе двоичного поиска в множестве из $n$ элементов, каждый из которых является целым числом в диапазоне от 1 до, например, $n$. Для представления каждого из этих целых чисел требуется $\lg(n^2) = 2\lg n$ битов, а поскольку количество чисел в множестве поиска равно $\lg n$, то общее количество битов будет равно $2\lg^2 n$.

  Функция «логарифм в квадрате» обычно возникает при разработке сложных гнездовых структур, где каждый узел в, скажем, двоичном дереве представляет другую структуру данных, возможно, упорядоченную по другому ключу.

- $f(n) = \sqrt{n}$.

  Функция квадратного корня встречается не так уж редко, но представляет класс «сублинейных полиномов», поскольку $\sqrt{n} = n^{1/2}$. Такие функции возникают при построении $d$-мерных сеток, содержащих $n$ точек. Площадь квадрата размером $\sqrt{n} \times \sqrt{n}$ равна $n$, а объем куба размером $n^{1/3} \times n^{1/3} \times n^{1/3}$ также составляет $n$. В общем, объем $d$-мерного гиперкуба со стороной $n^{1/d}$ составляет $n$.

- $f(n) = n^{(1+\varepsilon)}$.

  Греческая буква ε обозначает константу, которая может быть сколь угодно малой, но при этом не равняется нулю.

  Она может возникнуть в следующих обстоятельствах. Допустим, время исполнения алгоритма равно $2^c n^{(1 + 1/c)}$, и мы можем выбирать любое значение для $c$. При $c = 2$ время исполнения будет $4n^{3/2}$, или $O(n^{3/2})$. При $c = 3$ время исполнения будет $8n^{4/3}$, или $O(n^{4/3})$, что уже лучше. Действительно, чем больше значение $c$, тем лучше становится показатель степени.

  Но $c$ нельзя сделать как угодно большим, до того как член $2^c$ станет доминировать. Вместо этого мы обозначаем время исполнения этого алгоритма как $O(n^{1 + \varepsilon})$ и предоставляем пользователю определить наилучшее значение для ε.

## 2.10.2. Пределы и отношения доминирования

Отношения доминирования между функциями являются следствием теории пределов, изучаемой в курсе высшей математики. Говорят, что функция $f(n)$ *доминирует* над функцией $g(n)$, если $\lim_{n\to\infty} g(n)/f(n) = 0$.

Рассмотрим это определение в действии. Допустим, что $f(n) = 2n^2$ и $g(n) = n^2$. Очевидно, что $f(n) > g(n)$ для всех $n$, но не доминирует над ней, т. к.

$$\lim_{x\to\infty} \frac{g(n)}{f(n)} = \lim_{x\to\infty} \frac{n^2}{2n^2} = \lim_{x\to\infty} \frac{1}{2} \neq 0.$$

Этого следовало ожидать, т. к. обе функции принадлежат к одному и тому же классу $\Theta(n^2)$. Теперь рассмотрим функции $f(n) = n^3$ и $g(n) = n^2$. Так как

$$\lim_{x\to\infty} \frac{g(n)}{f(n)} = \lim_{x\to\infty} \frac{n^2}{n^3} = \lim_{x\to\infty} \frac{1}{n} = 0,$$

то доминирует многочлен более высокого уровня. Это справедливо для любых двух многочленов, т. е. $n^a$ доминирует над $n^b$, если $a > b$, т. к.

$$\lim_{x\to\infty} \frac{n^b}{n^a} = \lim_{x\to\infty} n^{b-a} \to 0.$$

Таким образом, $n^{1,2}$ доминирует над $n^{1,1999999}$.

Перейдем к показательным функциям: $f(n) = 3^n$ и $g(n) = 2^n$. Так как

$$\lim_{x\to\infty}\frac{g(n)}{f(n)} = \lim_{x\to\infty}\frac{2^n}{3^n} = \lim_{x\to\infty}\left(\frac{2}{3}\right)^n = 0,$$

значит, доминирует функция с большим основанием.

Возможность доказать отношение доминирования зависит от возможности доказать пределы. Давайте рассмотрим одну важную пару функций. Любой многочлен (скажем, $f(n) = n^\varepsilon$) доминирует над логарифмическими функциями (например, $g(n) = \lg n$). Так как $n = 2^{\lg n}$, то

$$f(n) = \left(2^{\lg n}\right)^\varepsilon = 2^{\varepsilon \lg n}.$$

Теперь рассмотрим следующее тождество:

$$\lim_{x\to\infty}\frac{g(n)}{f(n)} = \lg n / 2^{\varepsilon \lg n}.$$

В действительности при $n\to\infty$ оно стремится к нулю.

#### Подведение итогов

Рассматривая совместно функции, приведенные в этом разделе, и функции, обсуждаемые в *разд. 2.3.1*, мы видим полную картину порядка доминирования функций:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\varepsilon} \gg n\log n \gg n \gg \sqrt{n} \gg \log^2 n \gg \log n \gg \log n/\log\log n \gg \gg$$
$$\gg \gg \log\log n \gg \alpha(n) \gg 1.$$

# Замечания к главе

В других работах по алгоритмам уделяется значительно больше внимания формальному анализу алгоритмов, чем в этой книге, поэтому читатели со склонностью к теоретическим выкладкам отсылаются к другим изданиям. В частности, анализ алгоритмов рассматривается на более глубоком уровне в книгах [CLRS09] и [KT06].

В книге [GKP89] дается интересное и всестороннее изложение математического аппарата для анализа алгоритмов. В книге [NZM91] дается мое любимое введение в теорию чисел, включая проблему Уоринга, упомянутую в *разд. 2.8*.

Понятие доминирования также порождает обозначение «Little Oh» ($o$-малое). Говорят, что $f(n) = o(g(n))$ тогда и только тогда, когда $g(n)$ доминирует над $f(n)$. Среди прочего обозначение $o$-малое полезно для формулировки задач. Требование представить алгоритм с временной сложностью $o(n^2)$ означает, что нам нужен алгоритм с функцией временной сложности лучшей, чем квадратичная, и что будет приемлемой временная сложность, выражаемая функцией $O(n^{1,999}\log^2 n)$.

# 2.11. Упражнения

## Анализ программ

1. *[3]* Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
Mystery(n)
   r = 0
   for i = 1 to n - 1 do
      for j = i + 1 to n do
         for k = 1 to j do
           r = r + 1
   return(r)
```

2. *[3]* Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
Pesky(n)
 r = 0
 for i = 1 to n do
   for j = 1 to i do
     for k = j to i + j do
       r = r + 1
 return(r)
```

3. *[5]* Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
Pestiferous(n)
     r = 0
     for i = 1 to n do
       for j = 1 to i do
         for k = j to i + j do
           for l = 1 to i + j - k do
             r = r + 1
      return(r)
```

4. *[8]* Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
Conundrum(n)
 r = 0
 for i = 1 to n do
   for j = i + 1 to n do
     for k = i + j - 1 to n do
       r = r + 1
 return(r)
```

5. *[5]* Рассмотрим следующий алгоритм: (команда `print` отображает один символ звездочки (`*`), а команда `x =2x` удваивает значение переменной `x`):

```
for k = 1 to n:
    x = k
    while (x < n):
        print '*'
        x = 2x
```

Пусть $f(n)$ представляет временную сложность этого алгоритма (или, равнозначно, количество раз вывода на печать символа `*`). Предоставьте правильные границы для $O(f(n))$ и $\Omega(f(n))$, в идеале сходящиеся на $\Theta(f(n))$.

6. *[5]* Допустим, что для вычисления многочлена $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ используется следующий алгоритм:

```
  p = a0;
xpower = 1;
for i = 1 to n do
          xpower = x · xpower;
          p = p + ai * xpower
```

- Сколько операций умножения выполняется в наихудшем случае? А сколько операций сложения?
- Сколько операций умножения выполняется в среднем?
- Можно ли улучшить этот алгоритм?

7. *[3]* Докажите правильность следующего алгоритма для вычисления максимального значения в массиве $A[1..n]$:

```
max(A)
    m = A[1]
    for i = 2 to n do
            if A[i] > m then m = A[i]
    return (m)
```

## Упражнения по асимптотическим обозначениям

8. *[3]* Верно или неверно?

   a) $2^{n+1} = O(2^n)$;

   b) $2^{2n} = O(2^n)$.

9. *[3]* Для каждой из следующих пар функций функция $f(n)$ является членом одного из множеств функций $O(g(n))$, $\Omega(g(n))$ или $\Theta(g(n))$ Определите, членом какого множества является функция в каждом случае, и вкратце обоснуйте свой вывод:

   a) $f(n) = \log n^2$; $g(n) = \log n + 5$;

   b) $f(n) = \sqrt{n}$ ; $g(n) = \log n^2$;

   c) $f(n) = \log^2 n$; $g(n) = \log n$;

   d) $f(n) = n$; $g(n) = \log^2 n$;

   e) $f(n) = n \log n + n$; $g(n) = \log n$;

f) $f(n) = 10$; $g(n) = \log 10$;

g) $f(n) = 2^n$; $g(n) = 10n^2$;

h) $f(n) = 2^n$; $g(n) = 3^n$.

10. *[3]* Для каждой из следующих пар функций $f(n)$ и $g(n)$ определите, справедливы ли выражения: $f(n) = O(g(n))$ и $g(n) = O(f(n))$ или оба:

a) $f(n) = (n^2 - n)/2$; $g(n) = 6n$;

b) $f(n) = n + 2\sqrt{n}$ ; $g(n) = n^2$;

c) $f(n) = n \log n$; $g(n) = n\sqrt{n}/2$;

d) $f(n) = n + \log n$; $g(n) = \sqrt{n}$ ;

e) $f(n) = 2(\log n)^2$; $g(n) = \log n + 1$;

f) $f(n) = 4n \log n + n$; $g(n) = (n^2 - n)/2$.

11. *[5]* Какие асимптотические границы справедливы для следующих функций $f(n)$: $O(g(n))$, $\Omega(g(n))$ или $\Theta(g(n))$?

a) $f(n) = 3n^2$, $g(n) = n^2$;

b) $f(n) = 2n^4 - 3n^2 + 7$, $g(n) = n^5$;

c) $f(n) = \log n$, $g(n) = \log n + 1/n$;

d) $f(n) = 2^{k \log n}$, $g(n) = n^k$;

e) $f(n) = 2^n$, $g(n) = 2^{2n}$.

12. *[3]* Докажите, что $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

13. *[3]* Докажите, что $n^2 = O(2^n)$.

14. *[3]* Докажите или опровергните, что $\Theta(n^2) = \Theta(n^2 + 1)$.

15. *[3]* Предположим, что имеется пять алгоритмов с указанными далее временами исполнения. (Пусть это точные времена исполнения.) Насколько медленнее будут эти алгоритмы исполняться, если (*a*) удвоить размер ввода или (*b*) увеличить размер ввода на один элемент?

a) $n^2$;

b) $n^3$;

c) $100n^2$;

d) $n \log n$;

e) $2^n$.

16. *[3]* Предположим, что имеется шесть алгоритмов с указанными далее временами исполнения. (Допустим, что это точное количество операций, выполняемых как функция размера ввода $n$.) Также предположим наличие компьютера, способного выполнять $10^{10}$ операций в секунду. Какой самый больший размер ввода $n$ каждый алгоритм может выполнить в течение одного часа?

a) $n^2$;

b) $n^3$;

c) $100n^2$;

d) $n \log n$;

e) $2^n$ (f) $2^{2^n}$.

17. *[3]* Для каждой из следующих пар функций $f(n)$ и $g(n)$ найдите положительную константу $c$, при которой $f(n) \le c \cdot g(n)$ для всех $n > 1$:

a) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$;

b) $f(n) = n\sqrt{n} + n^2$, $g(n) = n^2$;

c) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$.

18. *[3]* Докажите, что если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, тогда $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

19. *[3]* Докажите, что если $f_1(n) = \Omega(g_1(n))$ и $f_2(n) = \Omega(g_2(n))$, тогда $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

20. *[3]* Докажите, что если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, тогда $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

21. *[5]* Докажите, что для всех $k \ge 0$ и всех множеств вещественных констант $\{a_k, a_{k-1}, ..., a_1, a_0\}$ верно: $a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = O(n^k)$.

22. *[5]* Докажите, что для любых вещественных констант $a$ и $b$, $b > 0$ верно $(n + a)^b = \Theta(n^b)$.

23. *[5]* Упорядочите приведенные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

| | | | |
|---|---|---|---|
| $n$ | $2^n$ | $n \lg n$ | $\ln n$ |
| $n - n^3 + 7n^5$ | $\lg n$ | $\sqrt{n}$ | $e^n$ |
| $n^2 + \lg n$ | $n^2$ | $2^{n-1}$ | $\lg \lg n$ |
| $n^3$ | $(\lg n)^2$ | $n!$ | $n^{1+\varepsilon}$, где $0 < \varepsilon < 1$ |

24. *[8]* Упорядочите приведенные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

| | | | |
|---|---|---|---|
| $n^\pi$ | $\pi^n$ | $\binom{n}{5}$ | $\sqrt{2^{\sqrt{n}}}$ |
| $\binom{n}{n-4}$ | $2^{\log^2 n}$ | $n^{5(\log n)^2}$ | $n^4\binom{n}{n-4}$ |

25. *[8]* Упорядочите приведенные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

| | | | |
|---|---|---|---|
| $\sum_{i=1}^{n} i^i$ | $n^n$ | $(\log n)^{\log n}$ | $2^{(\log n^2)}$ |
| $n!$ | $2^{\log^4 n}$ | $n^{(\log n)^2}$ | $\binom{n}{n-4}$ |

26. *[5]* Упорядочите приведенные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

| $\sqrt{n}$ | $n$ | $2^n$ |
|---|---|---|
| $n \log n$ | $n - n^3 + 7n^5$ | $n^2 + \log n$ |
| $n^2$ | $n^3$ | $\log n$ |
| $n^{1/3} + \log n$ | $(\log n)^2$ | $n!$ |
| $\ln n$ | $n/\log n$ | $\log \log n$ |
| $(1/3)^n$ | $(3/2)^n$ | 6 |

27. *[5]* Найдите две функции $f(n)$ и $g(n)$, которые удовлетворяют приведенным условиям. Если таких $f$ и $g$ нет, укажите на этот факт в ответе:

a) $f(n) = o(g(n))$ и $f(n) \neq \Theta(g(n))$;

b) $f(n) = \Theta(g(n))$ и $f(n) = o(g(n))$;

c) $f(n) = \Theta(g(n))$ и $f(n) \neq O(g(n))$;

d) $f(n) = \Omega(g(n))$ и $f(n) \neq O(g(n))$.

28. *[5]* Верно или неверно?

a) $2n^2 + 1 = O(n^2)$;

b) $\sqrt{n} = O(\log n)$;

c) $\log n = O(\sqrt{n})$;

d) $n^2(1 + \sqrt{n}) = O(n^2 \log n)$;

e) $3n^2 + \sqrt{n} = O(n^2)$;

f) $\sqrt{n} \log n = O(n)$;

g) $\log n = O(n^{-1/2})$.

29. *[5]* Для каждой из следующих пар функций $f(n)$ и $g(n)$ укажите, какие из следующих равенств справедливы: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$:

a) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$;

b) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$;

c) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$.

30. *[3]* Ответьте *да* или *нет* на следующие вопросы и вкратце обоснуйте свой ответ:

a) Если время исполнения алгоритма в наихудшем случае определяется как $O(n^2)$, возможно ли, что для некоторых входных экземпляров это время будет определяться как $O(n)$?

b) Если время исполнения алгоритма в наихудшем случае определяется как $O(n^2)$, возможно ли, что для всех входных экземпляров это время будет определяться как $O(n)$?

c) Если время исполнения алгоритма в наихудшем случае определяется как $\Theta(n^2)$, возможно ли, что на некоторых входных экземплярах это время будет определяться как $O(n)$?

d) Если время исполнения алгоритма в наихудшем случае определяется как $\Theta(n^2)$, возможно ли, что для всех входных экземпляров это время будет определяться как $O(n)$?

e) Принадлежит ли функция $f(n)$ множеству $\Theta(n^2)$ (т. е. $f(n) = \Theta(n^2)$), где $f(n) = 100n^2$ для четных $n$ и $f(n) = 20n^2 - n\log_2 n$ для нечетных $n$?

31. *[3]* Определите, верны или нет следующие тождества, либо укажите, что это определить невозможно. Обоснуйте свой ответ:

a) $3^n = O(2^n)$;

b) $\log 3^n = O(\log 2^n)$;

c) $3^n = \Omega(2^n)$;

d) $\log 3^n = \Omega(\log 2^n)$.

32. *[5]* Для каждой из следующих функций $f(n)$ найдите простую функцию $g(n)$, для которой $f(n) = \Theta(g(n))$:

a) $f(n) = \sum_{i=1}^{n} \frac{1}{i}$;

b) $f(n) = \sum_{i=1}^{n} \left\lceil \frac{1}{i} \right\rceil$;

c) $f(n) = \sum_{i=1}^{n} \log i$;

d) $f(n) = \log(n!)$.

33. *[5]* Расположите следующие функции в возрастающем порядке:

$$f_1(n) = n^2 \log_2 n,\ f_2(n) = n(\log_2 n)^2,\ f_3(n) = \sum_{i=0}^{n} 2^i,\ f_4(n) = \log_2\left(\sum_{i=0}^{n} 2^i\right).$$

34. *[5]* Какое из следующих уравнений верно?

a) $\sum_{i=1}^{n} 3^i = \Theta(3^{n-1})$;

b). $\sum_{i=1}^{n} 3^i = \Theta(3^n)$;

c) $\sum_{i=1}^{n} 3^i = \Theta(3^{n+1})$.

35. *[5]* Для каждой из следующих функций $f(n)$ найдите простую функцию $g(n)$, при которой $f(n) = \Theta(g(n))$.

a) $f_1(n) = (1000)2^n + 4^n$;

b) $f_2(n) = n + n\log n + \sqrt{n}$;

c) $f_3(n) = \log(n^{20}) + (\log n)^{10}$;

d) $f_4(n) = (0,99)^n + n^{100}$.

36. *[5]* Для каждой пары выражений $(A, B)$ в приведенной таблице укажите, какой именно функцией является $A$ для $B$ — функцией $O$, $o$, $\Omega$, $w$ или $\Theta$. Обратите внимание, что для любой из этих пар возможны несколько, один или ни одного варианта. Укажите все правильные отношения:

| | $A$ | $B$ |
|---|---|---|
| a) | $n^{100}$ | $2^n$ |
| b) | $(\lg n)^{12}$ | $\sqrt{n}$ |
| c) | $\sqrt{n}$ | $n^{\cos(\pi n/8)}$ |
| d) | $10^n$ | $100^n$ |
| e) | $n^{\lg n}$ | $(\lg n)^n$ |
| f) | $\lg(n!)$ | $n \lg n$ |

## Суммирование

37. *[5]* Определите формулу суммы для чисел строки $i$ следующего треугольника и докажите ее правильность. Каждый элемент строки является суммой части строки из трех элементов непосредственно над ним. Отсутствующие элементы полагаются равными нулю.

```
                1
              1   1   1
            1   2   3   2   1
          1   3   6   7   6   3   1
        1   4  10  16  19  16  10   4   1
```

38. *[3]* Допустим, что рождественские праздники длятся $n$ дней. Сколько точно подарков прислала мне моя «true love» (*«истинная любовь»*)? (Если вы не понимаете, о чем речь, выясните смысл вопроса самостоятельно.)

39. *[5]* Неупорядоченный массив размером $n$ содержит последовательность разных целых чисел от 1 до $n + 1$, в которой отсутствует один член. Создайте алгоритм с временной сложностью $O(n)$, чтобы определить отсутствующее целое число, при этом не используя никакого дополнительного пространства.

40. *[5]* Рассмотрим следующий фрагмент кода:

```
for i=1 to n do
   for j=i to 2*i do
      output "foobar"
```

Пусть $T(n)$ означает, сколько раз слово `"foobar"` печатается в зависимости от значения $n$.

a) Выразите $T(n)$ в виде суммы (точнее, в виде двух вложенных сумм).

b) Упростите выражение суммы. Полностью распишите все преобразования.

41. *[5]* Рассмотрим следующий фрагмент кода:

```
for i=1 to n/2 do
   for j=i to n-i do
      for k=1 to j  do
         output "foobar"
```

Допустим, что $n$ — четное. Пусть $T(n)$ означает, сколько раз слово `"foobar"` выводится в зависимости от значения $n$.

a) Выразите функцию $T(n)$ в виде трех вложенных сумм.

b) Упростите выражение суммирования. Полностью распишите все преобразования.

42. *[6]* На уроках арифметики в школе нам говорили, что $x \times y$ означает, что число $x$ нужно написать $y$ раз подряд и сосчитать сумму, т. е. $5 \times 4 = 5 + 5 + 5 + 5 = 20$. Выразите в виде функции от $n$ и $b$ временную сложность умножения двух чисел, которые в $b$-ичной системе счисления состоят из $n$ цифр (люди работают в десятичной системе счисления, а компьютеры — в двоичной), методом многократного сложения. Будем считать, что умножение или сложение однозначных чисел занимает $O(1)$ времени. (Подсказка: подумайте, насколько большим может быть число $y$ в зависимости от $n$ и $b$?)

43. *[6]* На уроках арифметики нас также учили умножать большие числа поразрядно, т. е. $127 \times 211 = 127 \times 1 + 127 \times 10 + 127 \times 200 = 26\,797$. Выразите в виде функции от $n$ временную сложность умножения этим методом двух чисел из $n$ цифр. Будем считать, что умножение или сложение однозначных чисел занимает $O(1)$ времени.

## Логарифмы

44. *[5]* Докажите следующие тождества:

    a) $\log_a(xy) = \log_a x + \log_b y$;

    b) $\log_a x^y = y \log_a x$;

    c) $\log_a x = \dfrac{\log_b x}{\log_b a}$;

    d) $x^{\log_b y} = y^{\log_b x}$.

45. *[3]* Докажите, что $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.

46. *[3]* Докажите, что двоичное представление числа $n \geq 1$ содержит $\lfloor \lg_2 n \rfloor + 1$ битов.

47. *[5]* В одной из моих научных статей я привел пример алгоритма сортировки методом сравнений с временной сложностью $O(n \log(\sqrt{n}))$. Почему это возможно, если нижняя граница сортировки определена как $\Omega(n \log n)$?

## Задачи, предлагаемые на собеседовании

48. *[5]* Имеется множество $S$ из $n$ чисел. Из этого множества нужно выбрать такое подмножество $S'$ из $k$ чисел, чтобы вероятность вхождения каждого элемента из множества $S$ в подмножество $S'$ была одинаковой (т. е. чтобы вероятность выбора каждого элемента была равна $k/n$). Выбор нужно сделать за один проход по числам множества $S$. Решите задачу также для случая, когда число $n$ неизвестно?

49. *[5]* Нужно сохранить тысячу элементов данных в тысяче узлов. В каждом узле можно сохранить копии только трех разных элементов. Разработайте схему создания копий, позволяющую минимизировать потерю данных при выходе узлов из строя. Сколько утерянных элементов данных нужно ожидать в случае выхода из строя трех произвольных узлов?

50. *[5]* Имеется следующий алгоритм поиска наименьшего числа в массиве чисел $A[0, ..., n]$. Для хранения текущего минимального числа используется переменная *tmp*. Начиная с ячейки массива $A[0]$, значение переменной *tmp* сравнивается по порядку со значениями ячеек $A[1]$, $A[2]$, ..., $A[N]$. Если значение ячейки $A[i]$ окажется меньше значения *tmp*

($A[i] < tmp$), то оно записывается в переменную $tmp$ ($tmp = A[i]$). Сколько нужно ожидать таких операций присваивания?

51. *[5]* Вам дали 10 кошельков с золотыми монетами. Монеты в девяти из этих кошельков весят по 10 грамм каждая, а в оставшемся кошельке — на 1 грамм меньше. С помощью цифровых весов нужно найти кошелек с легкими монетами, выполнив лишь одно взвешивание.

52. *[5]* У вас есть восемь шариков одинакового размера. Семь из них имеют одинаковый вес, восьмой шарик чуть тяжелее остальных. Нужно найти этот шарик, выполнив лишь два взвешивания.

53. *[5]* Допустим, что планируется слить $n$ компаний в одну. Сколько имеется разных способов для осуществления этого слияния?

54. *[7]* Шести пиратам нужно поделить между собой 300 долларов следующим способом. Самый старший пират предлагает, как нужно разделить деньги, после чего пираты голосуют по его предложению. Если предложение одобрено, по крайней мере половиной пиратов, то деньги распределяются в соответствии с предложенным способом. В противном случае автора предложения убивают, и следующий по старшинству пират предлагает свой способ. Процесс повторяется. Ответьте, каков будет результат этого деления, и обоснуйте. То есть сколько пиратов останется в живых и каким образом будут распределены деньги? Все пираты обладают хорошим умом, и самая приоритетная задача каждого — остаться в живых, а следующая по важности — получить как можно большую долю денег.

55. *[7]* Вариант предыдущей задачи. Пираты делят только один неделимый доллар. Кто получит этот доллар и сколько пиратов будет убито?

## LeetCode

1. **https://leetcode.com/problems/remove-k-digits/**

1. **https://leetcode.com/problems/counting-bits/**

2. **https://leetcode.com/problems/4sum/**

## HackerRank

1. **https://www.hackerrank.com/challenges/pangrams/**

2. **https://www.hackerrank.com/challenges/the-power-sum/**

3. **https://www.hackerrank.com/challenges/magic-square-forming/**

## Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**:

1. «Primary Arithmetic», глава 5, задача 10035.

2. «A Multiplication Game», глава 5, задача 847.

3. «Light, More Light», глава 7, задача 10110.

ГЛАВА 3

# Структуры данных

Вставку правильной структуры данных в медленно работающую программу можно сравнить с пересадкой органа. Такие важные классы *абстрактных типов данных*, как контейнеры, словари и очереди с приоритетами, могут реализовываться посредством разных, но функционально эквивалентных *структур данных*. Замена структуры данных одного типа структурой данных другого типа не влияет на правильность программы, поскольку предполагается, что одна правильная реализация заменяется другой правильной реализацией. Но в реализации другого типа данных могут применяться операции с иными временны́ми отношениями, в результате чего общая производительность программы может значительно повыситься. Подобно ситуации с больным, нуждающемся в пересадке одного органа, для повышения производительности программы может оказаться достаточным заменить лишь один ее компонент.

Конечно, лучше с рождения иметь здоровое сердце, чем жить в ожидании донорского. То же самое справедливо и в случае со структурами данных. Наибольшую пользу от применения правильных структур данных можно получить, лишь заложив их использование в программу с самого начала. При подготовке этой книги предполагалось, что ее читатели уже имеют знания об элементарных структурах данных и манипуляциях с указателями. Но т. к. в сегодняшних курсах по структурам данных внимание фокусируется больше на абстракции данных и на объектно-ориентированном программировании, чем на деталях представления структур в памяти, то мы повторим этот материал здесь, чтобы убедиться в том, что вы его полностью понимаете.

Как и при изучении большинства предметов, при изучении структур данных важнее хорошо освоить основной материал, чем бегло ознакомиться с более сложными понятиями. Здесь мы обсудим три фундаментальных абстрактных типа данных, *контейнеры*, *словари* и *очереди с приоритетами*, и рассмотрим, как они реализуются посредством массивов и списков. Более сложные реализации структур данных рассматриваются в релевантной задаче в каталоге задач.

## 3.1. Смежные и связные структуры данных

В зависимости от реализации (посредством массивов или указателей) структуры данных можно четко разбить на два типа: *смежные* и *связные*.

- *Смежные структуры данных* реализованы в виде непрерывных блоков памяти. К ним относятся массивы, матрицы, кучи и хеш-таблицы.
- *Связные структуры данных* реализованы в отдельных блоках памяти, связанных вместе с помощью *указателей*. К этому виду структур данных относятся списки, деревья и списки смежных вершин графов.

В этом разделе дается краткий обзор сравнительных характеристик смежных и связных структур данных. Разница между ними более тонкая, чем может показаться на первый взгляд, поэтому я призываю не игнорировать этот материал, даже если вы и знакомы с этими типами структур данных.

### 3.1.1. Массивы

*Массив* представляет собой основную структуру данных смежного типа. Записи данных в массивах имеют постоянный размер, что позволяет с легкостью найти любой элемент по его *индексу* (или адресу).

Хорошей аналогией массива будет улица с домами, где каждый элемент массива соответствует дому, а индекс элемента — номеру дома. Считая, что все дома одинакового размера и пронумерованы последовательно от 1 до $n$, можно определить точное местонахождение каждого дома по его адресу[1].

Приведем достоинства массивов:

- *Постоянное время доступа при условии наличия индекса.*

  Так как индекс каждого элемента массива соответствует определенному адресу в памяти, то при наличии соответствующего индекса доступ к произвольному элементу массива осуществляется практически мгновенно.

- *Эффективное использование памяти.*

  Массивы содержат только данные, поэтому память не тратится на указатели и другую форматирующую информацию. Кроме этого, для элементов массива не требуется использовать метку конца записи, поскольку все элементы массива имеют одинаковый размер.

- *Локальность в памяти.*

  Во многих задачах программирования требуется обрабатывать элементы структуры данных в цикле. Массивы хорошо подходят для операций такого типа, поскольку обладают отличной локальностью в памяти. В современных компьютерных архитектурах физическая непрерывность последовательных обращений к данным помогает воспользоваться высокоскоростной кэш-памятью.

Недостатком массивов является то, что их размер нельзя изменять в процессе исполнения программы. Попытка обращения к $(n + 1)$-му элементу массива размером $n$ элементов немедленно вызовет аварийное завершение программы. Этот недостаток можно компенсировать объявлением массивов очень больших размеров, но это может повлечь за собой чрезмерные затраты памяти, что опять наложит ограничения на возможности программы.

В действительности, размеры массива можно изменять во время исполнения программы посредством приема, называющегося *динамическим выделением памяти*. Допустим, мы начнем с одноэлементного массива размером $m$ и будем удваивать его каждый

[1] Эта аналогия неприменима в случае с нумерацией домов в Японии. Там дома нумеруют в порядке их возведения, а не физического расположения. Поэтому найти какой-либо дом в Японии по его адресу, не имея карты, очень трудно.

раз до $2m$, когда предыдущий размер становится недостаточным. Этот процесс состоит из выделения памяти под новый непрерывный массив размером $2m$, копирования содержимого старого массива в нижнюю половину нового и возвращения памяти старого массива в систему распределения памяти.

Очевидным расточительством в этой процедуре является операция копирования содержимого старого массива в новый при каждом удвоении размера массива. Это порождает вопрос: какой объем работы нужно на самом деле выполнить? Чтобы получить в массиве $n$ позиций, потребуется $\log 2n$ (или $\lg n$) удвоений, плюс еще одно последнее удвоение на последней вставке, когда $n = 2j$ для некоторых значений $j$. Операции повторного копирования выполняются после первой, второй, четвертой, восьмой, ..., $n$-й вставок. На $i$-м удвоении количество операций копирования будет $2i - 1$, поэтому общее количество перемещений $M$ будет равно:

$$M = n + \sum_{i=1}^{\lg n} 2^{i-1} = 1 + 2 + 4 + \cdots + \frac{n}{2} + n = \sum_{i=1}^{\lg n} \frac{1}{2^i} = 2n.$$

Таким образом, каждый из $n$ элементов массива в среднем перемещается только два раза, а общая временная сложность управления динамическим массивом определяется той же самой функцией $O(n)$, какая справедлива для работы с одним статическим массивом достаточного размера.

Самой главной проблемой при использовании динамических массивов является отсутствие гарантии постоянства времени выполнения каждой вставки *в наихудшем случае*. Теперь же все обращения и *большинство* вставок будут быстрыми, за исключением тех относительно нечастых вставок, вызывающих удвоение массива. Зато у нас есть уверенность, что вставка $n$-го элемента в массив будет выполнена достаточно быстро, чтобы *общее* затраченное усилие осталось таким же $O(n)$. Такие *амортизированные* гарантии часто возникают при анализе структур данных.

## 3.1.2. Указатели и связные структуры данных

*Указатели* позволяют удерживать воедино связные структуры. Указатель — это адрес ячейки памяти. Переменная, содержащая указатель на элемент данных, может предоставить бóльшую гибкость в работе с этими данными, чем просто их копия. В качестве примера указателя можно привести номер сотового телефона, который позволяет связаться с владельцем телефона независимо от его местоположения внутри зоны действия сети.

Так как синтаксис и возможности указателей значительно различаются в разных языках программирования, то мы начнем с краткого обзора указателей в языке C. Переменная указателя `p` содержит адрес памяти, по которому находится определенный блок данных[2]. В языке C указатель при объявлении получает тип, соответствующий типу данных, на которые этот указатель может ссылаться. Операция, обозначаемая `*p`, называется *разыменованием указателя* и возвращает значение элемента данных, рас-

[2] В языке C разрешается прямая манипуляция адресами такими способами, которые могут привести Java-программистов в ужас, но в этой книге мы воздержимся от применения подобных методов.

положенного по адресу, содержащемуся в переменной указателя p. А операция, обозначаемая &x, называется *взятием адреса* и возвращает адрес (т. е. указатель) этой переменной x. Специальное значение NULL применяется для обозначения неинициализированного указателя или указателя на последний элемент структуры данных.

Все связные структуры данных имеют определенные общие свойства, что видно из следующего объявления типа связного списка (листинг 3.1).

**Листинг 3.1. Объявление структуры связного списка**

```
typedef struct list {
   item_type item;      /* Данные */
   struct list *next;   /* Указатель на следующий узел */
} list;
```

В частности:

- каждый узел в нашей структуре данных (структура list) содержит одно или несколько полей, предназначенных для хранения данных (поле item);
- каждый узел также содержит поле указателя на следующий узел (поле next). Это означает, что в связных структурах данных большой объем используемой ими памяти отдается под хранение указателей, а не полезных данных;
- наконец, требуется указатель на начало структуры, чтобы мы знали, откуда начинать обращение к ней.

Простейшей связной структурой является список, пример которого показан на рис. 3.1.

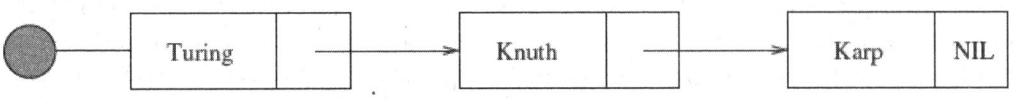



**Рис. 3.1.** Пример связного списка с полями данных и указателями

Списки поддерживают три основные операции: *поиск* (search), *вставку* (insert) и *удаление* (delete). В *двунаправленных* или *двусвязных* списках (doubly-linked list) каждый узел содержит указатель как на следующий, так и на предыдущий узел. Это упрощает определенные операции за счет дополнительного поля указателя в каждом узле.

## Поиск элемента в связном списке

Поиск элемента x в связном списке можно выполнять итеративным или рекурсивным методом. В листинге 3.2 приведен пример реализации рекурсивного поиска.

**Листинг 3.2. Рекурсивный поиск элемента в связном списке**

```
list *search_list(list *l, item_type x) {
    if (l == NULL) {
        return(NULL);
    }
```

```
    if (l->item == x) {
        return(l);
    } else {
        return(search_list(l->next, x));
    }
}
```

Если список содержит элемент x, то он находится либо в начале списка, либо в оставшейся части списка. В конце концов, задача сводится к поиску в пустом списке, который, очевидно, не может содержать элемент x.

## Вставка элемента в связный список

Код вставки элементов в однонаправленный связный список приведен в листинге 3.3.

**Листинг 3.3. Вставка элемента в однонаправленный связный список**

```
void insert_list(list **l, item_type x)
{
   list *p;          /* Временный указатель */
   p = malloc( sizeof(list) );
   p->item = x;
   p->next = *l;
   *l = p;
}
```

Поскольку нам не требуется содержать элементы списка в каком-либо определенном порядке, то мы можем вставлять каждый новый элемент туда, куда его удобнее всего вставить. Вставка элемента в начало списка позволяет избежать необходимости обхода списка, хотя и требует обновления указателя (переменная `l`) на начало списка.

Обратите внимание на две особенности языка C. Функция `malloc` возвращает указатель на блок памяти достаточного размера, выделенный для нового узла, в котором будет храниться элемент x. Двойная звездочка (`**l`) означает, что переменная `l` является *указателем на указатель* на узел списка. Таким образом, строчка кода `*l=p;` копирует значение `p` в блок памяти, на который ссылается переменная `l`, являющаяся внешней переменной, обеспечивающей доступ к началу списка.

## Удаление элемента из связного списка

Удаление элемента из связного списка является более сложной операцией. Сначала нам нужно найти указатель на элемент списка, *предшествующий* удаляемому элементу. Это выполняется рекурсивным способом (листинг 3.4).

**Листинг 3.4. Поиск указателя на элемент, предшествующий удаляемому**

```
list *item_ahead(list *l, list *x) {
    if ((l == NULL) || (l->next == NULL)) {
        return(NULL);
    }
```

```
    if ((l->next) == x) {
         return(l);
    } else {
        return(item_ahead(l->next, x));
    }
}
```

Найти элемент, предшествующий удаляемому, нужно потому, что он содержит указатель `next` на следующий, в нашем случае удаляемый, узел, который нужно обновить после удаления узла. После проверки существования узла, подлежащего удалению, собственно операция удаления не представляет собой ничего сложного. При удалении первого элемента связного списка нужно быть особенно внимательным, чтобы не забыть обновить указатель (переменную `l`) на начало списка (листинг 3.5).

**Листинг 3.5. Удаление элемента связного списка**

```
void delete_list(list **l, list **x) {
    list *p; /* Указатель на узель*/
    list *pred; /* Указатель на предшествующий узел*/

    p = *l;
    pred = item_ahead(*l, *x);

    if (pred == NULL) { /* Соединяем список */
        *l = p->next;
    } else {
        pred->next = (*x)->next;
    }
    free(*x); /* Освобождаем память узла */
}
```

В языке C требуется явное освобождение памяти, поэтому после удаления узла необходимо освободить занимаемую им память, чтобы возвратить ее в систему. В результате входящий указатель становится *висящим*, т. е. указывает на объект, которого больше нет. Поэтому необходимо быть внимательным, чтобы больше не задействовать этот указатель. Такие проблемы обычно не возникают в Java, благодаря использованию в этом языке более эффективной модели управления памятью.

## 3.1.3. Сравнение

Связные структуры имеют следующие преимущества над статическими массивами:

- переполнение в связных структурах невозможно, если только сама память не переполнена;
- операции вставки и удаления элементов *проще* соответствующих операций в статических массивах;
- при работе с большими записями перемещение указателей происходит легче и быстрее, чем перемещение самих записей.

И наоборот, относительные преимущества массивов таковы:

- эффективное использование памяти — связным структурам необходимо дополнительная память для хранения указателей;
- эффективный произвольный доступ к элементам массива;
- массивы обладают лучшей локальностью в памяти и более эффективны в использовании кэш-памяти, чем связные списки.

> ***Подведение итогов***
>
> Динамическое выделение памяти обеспечивает гибкость в выборе способа и момента использования этого ограниченного ресурса.

Напоследок заметим, что как связные списки, так и массивы можно рассматривать, как рекурсивные объекты:

- *Списки.*

  После удаления первого элемента связного списка мы имеем такой же связный список, только меньшего размера. То же самое справедливо для строк, поскольку в результате удаления символов из строки получается более короткая строка.

- *Массивы.*

  Отделение первых $k$ элементов от массива из $n$ элементов дает нам два массива меньших размеров, а именно размером $k$ и $n - k$ элементов соответственно.

Знание этого свойства позволяет упростить обработку списков и создавать эффективные алгоритмы типа «разделяй и властвуй», такие как быстрая сортировка (quicksort) и двоичный поиск.

## 3.2. Стеки и очереди

Термин *контейнер* (container) обозначает абстрактный тип данных, позволяющий хранить и извлекать данные *независимо от содержимого*. В противоположность контейнерам словари представляют собой абстрактные типы данных, которые извлекаются по ключевому значению или содержимому. Словари рассматриваются в *разд. 3.3*.

Контейнеры различаются по поддерживаемому ими типу извлечения данных. В двух наиболее важных типах контейнеров порядок извлечения зависит от порядка помещения:

- *Стеки.*

  Извлечение данных осуществляется в порядке LIFO («last-in, first-out», «последним вошел — первым вышел»). Стеки легко реализуются и обладают высокой эффективностью. По этой причине их удобно применять в случаях, когда порядок извлечения данных не имеет никакого значения, — например, при обработке пакетных заданий. Операции вставки и извлечения данных для стеков называются `push` (запись в стек) и `pop` (снятие со стека):

  - `push(x,s)` — вставить элемент `x` на верх (в конец) стека `s`;
  - `pop(s)` — извлечь (и удалить) верхний (последний) элемент из стека `s`.

Порядок LIFO возникает во многих реальных ситуациях. Например, из набитого битком вагона метро пассажиры выходят в порядке LIFO. Продукты из холодильника нередко вынимаются в этом же порядке, с игнорированием сроков годности. По крайней мере, такой порядок применяется в моем холодильнике. В алгоритмах порядок LIFO обычно возникает при выполнении рекурсивных операций.

- *Очереди.*

  Очереди поддерживают порядок извлечения FIFO («first-in, first-out», «первым вошел — первым вышел»). Использование этого порядка — определенно самый справедливый способ управления временем ожидания обслуживания. Обработка задач в порядке FIFO минимизирует максимальное время ожидания. Обратите внимание, что среднее время ожидания будет одинаковым, независимо от применяемого порядка, будь то FIFO или LIFO. Но, т. к. данные многих вычислительных приложений не теряют актуальность бесконечно долго, вопрос максимального времени ожидания становится чисто академическим.

  Очереди реализовать несколько труднее, чем стеки, и поэтому их применение больше подходит для приложений, в которых порядок извлечения данных является важным, — например, при эмуляции определенных процессов. Применительно к очередям операции вставки и извлечения данных называются `enqueue` (поставить в очередь) и `dequeue` (вывести из очереди) соответственно:

  - `enqueue(x,q)` — вставить элемент `x` в конец очереди `q`;
  - `dequeue(q)` — извлечь (и удалить) элемент в начале очереди `q`.

  Далее в книге мы увидим применение очередей в качестве основной структуры данных для управления поиском в ширину (BFS, breadth-first search) в графах.

На практике стеки и очереди можно реализовать посредством массивов или связных списков. Ключевой вопрос состоит в том, известна ли верхняя граница размера контейнера заранее, т. е. имеется ли возможность использовать статические массивы.

# 3.3. Словари

Тип данных *словарь* позволяет доступ к данным по содержимому. Словари применяются для хранения данных, которые можно быстро найти в случае надобности. Далее приводится список основных операций, поддерживаемых в словарях:

- `search(D,k)` — возвращает указатель на элемент словаря `D` с ключом `k`, если такой элемент существует;
- `insert(D,x)` — добавляет элемент, на который указывает `x`, в словарь `D`;
- `delete(D,x)` — удаляет из словаря `D` элемент, на который указывает `x`.

Некоторые словари также поддерживают другие полезные операции:

- `max(D)` и `min(D)` — возвращают указатель на элемент множества `D`, имеющий наибольший или наименьший ключ. Это позволяет использовать словарь в качестве очереди с приоритетами, которая рассматривается в *разд. 3.5*;
- `predecessor(D,x)` и `successor(D,x)` — возвращают элемент из отсортированного массива `D`, предшествующий элементу с ключом `k` или стоящий сразу после него со-

ответственно. Это позволяет обрабатывать в цикле все элементы этой структуры данных в отсортированном виде.

С помощью только что отмеченных словарных операций можно выполнять многие распространенные задачи обработки данных. Допустим, нам нужно удалить все повторяющиеся имена из списка рассылки, затем отсортировать и распечатать получившийся список. Для выполнения этой задачи инициализируем пустой словарь `D`, для которого ключом поиска будет служить имя записи. Потом считываем записи из списка рассылки и для каждой записи выполняем операцию `search` в словаре на предмет наличия в нем этого имени. Если имя отсутствует в словаре, то вставляем его с помощью операции `insert`. Обработав весь список, выводим на печать имена в словаре. Для этого, начиная с наименьшего элемента словаря (операция `min(D)`), выполняем операцию `successor`, пока не дойдем до наибольшего элемента (операция `max(D)`), обойдя таким образом по порядку все элементы словаря.

Определяя подобные задачи в терминах абстрактных операций над словарем, мы можем игнорировать детали представления структуры данных и концентрироваться на решении задачи.

Далее в этом разделе мы внимательно рассмотрим простые реализации словаря на основе массивов и связных списков. Более мощные реализации, такие как использование двоичных деревьев поиска (см. *разд. 3.4*) и хеш-таблицы (см. *разд. 3.7*), также являются привлекательными вариантами. Подробно словарные структуры данных рассматриваются в *разд. 15.1*. Читателям настоятельно рекомендуется просмотреть этот раздел, чтобы получить лучшее представление об имеющихся возможностях.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Сравнение реализаций словаря (I)

**ЗАДАЧА**. Определите асимптотическое время исполнения в наихудшем случае для всех семи основных словарных операций (`search`, `insert`, `delete`, `successor`, `predecessor`, `minimum` и `maximum`) для структуры данных, реализованной в виде:

- неотсортированного массива;
- отсортированного массива.

**РЕШЕНИЕ**. Эта (а также следующая) задача демонстрирует компромиссы, на которые приходится идти при разработке структур данных. Конкретное представление данных может обеспечить эффективную реализацию одних операций за счет снижения эффективности других.

В решении этой задачи мы будем полагать, что кроме массивов, упомянутых в формулировке, мы имеем доступ к нескольким переменным, в частности к переменной `n`, содержащей текущее количество элементов массива. Обратите внимание на необходимость обновлять эти переменные в операциях, изменяющих их значения (например, `insert` и `delete`), и на то, что стоимость такого сопровождения нужно включать в стоимость этих операций.

Сложность основных словарных операций для неотсортированных и отсортированных массивов показана в табл. 3.1. Звездочкой помечена операция, время исполнения которой можно сократить, применив оригинальный подход.

**Таблица 3.1.** Сложность основных словарных операций для массивов

| Словарная операция | Неотсортированный массив | Отсортированный массив |
|---|---|---|
| `search(A,k)` | $O(n)$ | $O(\log n)$ |
| `insert(A,x)` | $O(1)$ | $O(n)$ |
| `delete(A,x)` | $O(1)$* | $O(n)$ |
| `successor(A,x)` | $O(n)$ | $O(1)$ |
| `predecessor(A,x)` | $O(n)$ | $O(1)$ |

Чтобы понять, почему операция имеет такую сложность, необходимо выяснить, каким образом она реализуется. Рассмотрим сначала эти операции для неотсортированного массива $A$.

- При выполнении операции `search` ключ поиска `k` сравнивается (возможно) с каждым элементом неотсортированного массива. Таким образом, в наихудшем случае, когда ключ `k` отсутствует в массиве, время поиска будет линейным.
- При выполнении операции `insert` значение переменной $n$ увеличивается на единицу, после чего элемент $x$ копируется в $n$-ю ячейку массива $A[n]$. Основная часть массива не затрагивается этой операцией, поэтому время ее исполнения будет постоянным.
- Операция `delete` несколько сложнее, что обозначается звездочкой (*) в табл. 3.1. По определению в этой операции передается указатель $x$ на элемент, который нужно удалить, поэтому нам не нужно тратить время на поиск этого элемента. Но удаление элемента из массива оставляет в нем промежуток, который надо заполнить. Этот промежуток можно было бы заполнить, сдвинув все элементы массива, следующие за ним, т. е. элементы с $A[x + 1]$ по $A[n]$, вверх на одну позицию (т. е. по направлению к началу массива, уменьшая адрес каждого элемента на 1), но в случае удаления первого элемента эта операция займет время $\Theta(n)$. Поэтому будет намного лучше просто записать в ячейку $A[x]$ содержимое последней ячейки $A[n]$ массива и уменьшить значение $n$ на единицу. Время исполнения этой операции будет постоянным.
- Определения операций `predecessor` и `successor` даются для отсортированного массива. Но результатом выполнения этих операций в неотсортированном массиве не будет элемент $A[x - 1]$ (или $A[x + 1]$), поскольку физически предыдущий (или следующий элемент) не обязательно будет таковым логически. В нашем случае элементом, предшествующим элементу $A[x]$, будет наибольший из элементов, меньших $A[x]$. Аналогично следующим элементом за $A[x]$ будет наименьший из элементов, бóльших $A[x]$. Чтобы найти предшествующий или следующий элемент в неотсортированном массиве $A$, необходимо просмотреть все его $n$ элементов, что занимает линейное время.
- Операции `minimum` и `maximum` также определены только для отсортированного массива. Поэтому, чтобы найти наибольший или наименьший элемент в неотсортированном массиве, необходимо просмотреть все его элементы, что также занимает ли-

нейное время. Можно поддаться соблазну зарезервировать дополнительные переменные, содержащие текущие минимальное и максимальное значения, чтобы информацию об этих значениях можно было предоставить за время $O(1)$. Но это будет несовместимо с удалением за постоянное время, поскольку удалению элемента с минимальным значением сопутствует поиск нового минимума, занимающий линейное время.

Реализация словаря в отсортированном массиве переворачивает наши представления о трудных и легких операциях. В этом случае нахождение нужного элемента посредством двоичного поиска занимает время $O(\log n)$, т. к. мы знаем, что средний элемент находится в ячейке массива $A[n/2]$. А поскольку верхняя и нижняя половины массива также отсортированы, то поиск может продолжаться рекурсивно в соответствующей половине. Количество делений массива пополам, требуемое для получения половины из одного элемента, равно $[\lg n]$.

Отсортированность массива полезна и для других словарных операций. Наименьший и наибольший элементы находятся в ячейках $A[1]$ и $A[n]$ соответственно, а элементы, идущие непосредственно до и после элемента $A[x]$, — в ячейках $A[x-1]$ и $A[x+1]$ соответственно.

Но операции вставки и удаления элементов становятся более трудоемкими, поскольку создание места для нового элемента или заполнение промежутка после удаления может потребовать перемещения многих элементов массива. Вследствие этого обе операции исполняются за линейное время. ■

#### Подведение итогов

При разработке структуры данных должны быть сбалансированы скорости выполнения всех поддерживаемых ею операций. Структура, обеспечивающая максимально быстрое выполнение как операции $A$, так и операции $B$, вполне может оказаться не самой подходящей (в смысле скорости работы) для этих операций, взятых в отдельности.

### Остановка для размышлений: Сравнение реализаций словаря (II)

**ЗАДАЧА**. Определите асимптотическое время исполнения в наихудшем случае каждой из семи основных словарных операций для структуры данных, реализованной в виде:

- однонаправленного связного неотсортированного списка;
- двунаправленного связного неотсортированного списка;
- однонаправленного связного отсортированного списка;
- двунаправленного связного отсортированного списка.

**РЕШЕНИЕ**. При оценке производительности необходимо принимать во внимание два момента: тип связности списка (однонаправленная или двунаправленная) и его упорядоченность (отсортированный или неотсортированный). Производительность этих операций для каждого типа структуры данных показана в табл. 3.2. Операции, оценка производительности которых представляет особые трудности, помечены звездочкой (*).

**Таблица 3.2.** Производительность словарных операций в связных структурах данных

| Словарная операция | Односвязный неотсортированный список | Односвязный отсортированный список | Двусвязный неотсортированный список | Двусвязный отсортированный список |
|---|---|---|---|---|
| `search(L,sk)` | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| `insert(L,x)` | $O(1)$ | $O(n)$ | $O(1)$ | $O(n)$ |
| `delete(L,x)` | $O(n)$* | $O(n)$* | $O(1)$ | $O(1)$ |
| `successor(L,x)` | $O(n)$ | $O(1)$ | $O(n)$ | $O(1)$ |
| `predecessor(L,x)` | $O(n)$ | $O(n)$* | $O(n)$ | $O(1)$ |
| `minimum(L)` | $O(n)$ | $O(1)$ | $O(n)$ | $O(1)$ |
| `maximum(L)` | $O(n)$ | $O(1)$* | $O(n)$ | $O(1)$ |

Так же как и в случае с неотсортированными массивами, операция поиска неизбежно будет медленной, а операции модифицирования — быстрыми.

- Операции `insert`/`delete`. Здесь сложность возникает при удалении элемента из односвязного списка. По определению операции `delete` передается указатель $x$ на элемент, который нужно удалить (рис. 3.2). Однако в действительности нам нужен указатель на предшествующий элемент, поскольку именно его поле указателя необходимо обновить после удаления. Следовательно, требуется найти этот элемент, а его поиск в односвязном списке занимает линейное время[3]. Эта проблема не воз-

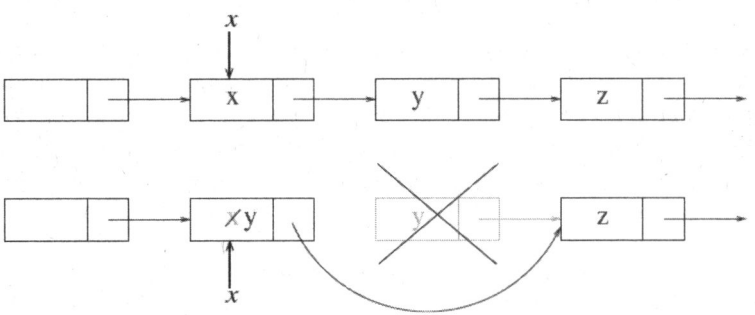



**Рис. 3.2.** Элемент можно удалить без обращения к предшествующему ему элементу списка, перезаписав его содержимое и удалив следующий за ним первоначально элемент

[3] Вообще-то, элемент из односвязного списка *можно* удалить за постоянное время, как показано на рис. 3.2. Для этого нужно в элемент, на который указывает $x$, записать содержимое элемента, на который указывает *x.next*, а затем освободить элемент, на который изначально указывал *x.next*. Нужно проявлять особую осторожность, когда $x$ указывает на первый или последний элемент списка (для этого используется постоянный сигнальный элемент, который всегда является последним в списке). Но это не позволит нам получить постоянное время выполнения операций поиска минимального/максимального значения, так как после удаления у нас больше не будет времени для нахождения новых конечных элементов.

никает в двусвязном списке, поскольку мы можем сразу же получить предшественника удаляемого элемента $x$.

Удаление элемента из отсортированного двусвязного списка выполняется быстрее, чем из отсортированного массива, поскольку связать предшествующий и последующий элементы списка дешевле, чем заполнять промежуток в массиве, перемещая оставшиеся элементы. Но удаление из односвязного отсортированного списка тоже усложняется необходимостью поиска предшествующего элемента.

- Операция `search`. Упорядоченность элементов не дает таких преимуществ при поиске в связном списке, как при поиске в массиве. Мы больше не можем выполнять двоичный поиск, поскольку не имеем возможности получить доступ к среднему элементу, не обратившись ко всем предшествующим ему элементам. Однако упорядоченность приносит определенную пользу в виде быстрого завершения неуспешного поиска. Действительно, если мы не нашли запись `Abbott`, дойдя до `Costello`[4], то можно сделать вывод, что такой записи в списке не существует. Тем не менее поиск в наихудшем случае занимает линейное время.

- Операции `predecessor` и `successor`. Реализация операции `predecessor` в односвязных списках усложняется уже упомянутой проблемой поиска указателя на предшествующий элемент. В отсортированных списках обоих типов логически следующий элемент эквивалентен следующему узлу, и поэтому время обращения к нему постоянно.

- Операция `maximum`. Наименьший элемент находится в начале отсортированного списка, и поэтому его легко найти. А наибольший элемент находится в конце списка, и чтобы добраться до него, обычно потребуется время $\Theta(n)$ как в односвязном, так и в двусвязном списке.

  Но можно поддерживать указатель на конец списка при условии, что нас устраивают расходы по обновлению этого указателя при каждой вставке и удалении. В двусвязных списках этот указатель на конечный элемент можно обновлять за постоянное время: при вставке проверять, имеет ли *last→next* значение `NULL`, а при удалении переводить указатель *last* на предшественника *last* в списке, если удален последний элемент.

  В случае односвязных списков у нас нет эффективного способа найти этого предшественника. Как же мы получаем время исполнения операции `maximum` равным $O(1)$? Трюк заключается в списывании затрат на каждое удаление, которое уже заняло линейное время. Дополнительный проход для обновления указателя, выполняемый за линейное время, не причиняет никакого вреда асимптотической сложности операции `delete`, но дает нам выигрыш в виде постоянного времени выполнения операции `maximum` (а также и операции `minimum`) как вознаграждение за четкое мышление. ■

---

[4] Abbott и Costello — знаменитый американский комический дуэт. — *Прим. пер.*

# 3.4. Двоичные деревья поиска

Пока что мы ознакомились со структурами данных, позволяющими выполнять быстрый поиск *или* гибкое обновление, но не быстрый поиск *и* гибкое обновление одновременно. В неотсортированных двусвязных списках для операций вставки и удаления записей требуется время $O(1)$, а для операций поиска — линейное время в худшем случае. Отсортированные массивы обеспечивают логарифмическое время выполнения двоичного поиска и линейное время выполнения вставок и удалений.

Для двоичного поиска необходимо иметь быстрый доступ к двум элементам — а именно: средним элементам верхней и нижней частей массива по отношению к заданному элементу. Иными словами, нужен *связный список*, в котором каждый узел содержит два указателя. Так мы приходим к идее двоичных деревьев поиска.

*Корневое двоичное дерево* определяется рекурсивно либо как пустое, либо как состоящее из узла, называющегося *корнем*, из которого выходят два корневых двоичных дерева, называющиеся *левым* и *правым поддеревом*. В корневых деревьях порядок «родственных» узлов имеет значение, т. е. левое поддерево отличается от правого. На рис. 3.3 приведены пять разных двоичных деревьев, которые можно создать из трех узлов.

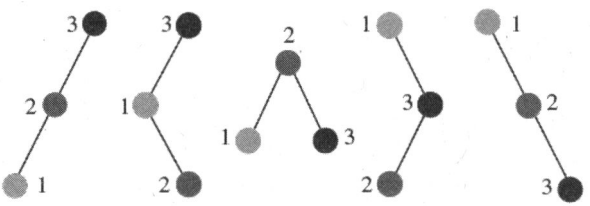



**Рис. 3.3.** Пять разных двоичных деревьев поиска с тремя узлами. Ключи всех элементов в левом поддереве узла $x$ имеют значение $< x$ (а в правом — $> x$)

Каждый узел двоичного *дерева поиска* помечается ключом $x$ таким образом, что для любого такого узла ключи всех узлов в его левом поддереве меньше $x$, а ключи всех узлов в его правом поддереве больше $x$[5]. Это весьма необычный способ постановки меток на деревьях поиска. Для любого двоичного дерева с количеством узлов $n$ и любого множества из $n$ ключей существует *только один* способ постановки меток, который делает его двоичным деревом поиска. Допустимые способы постановки меток для дерева двоичного поиска из трех узлов показаны на рис. 3.3.

## 3.4.1. Реализация двоичных деревьев

Кроме поля ключа, каждый узел двоичного дерева имеет поля `left`, `right` и `parent`, которые содержат указатели на левый и правый дочерние узлы и на родительский узел

[5] Дублирование ключей в деревьях двоичного поиска (или в любой другой словарной структуре) является плохой практикой, часто вызывающей чрезвычайно трудноуловимые ошибки. Лучшую поддержку повторяемых элементов можно организовать, добавив к каждому элементу третий указатель, явно поддерживая список всех элементов с этим ключом.

соответственно (единственным узлом, указатель `parent` которого равен `NULL`, является корневой узел всего дерева). Эти отношения показаны на рис. 3.4.

В листинге 3.6 приводится объявление типа для структуры дерева.

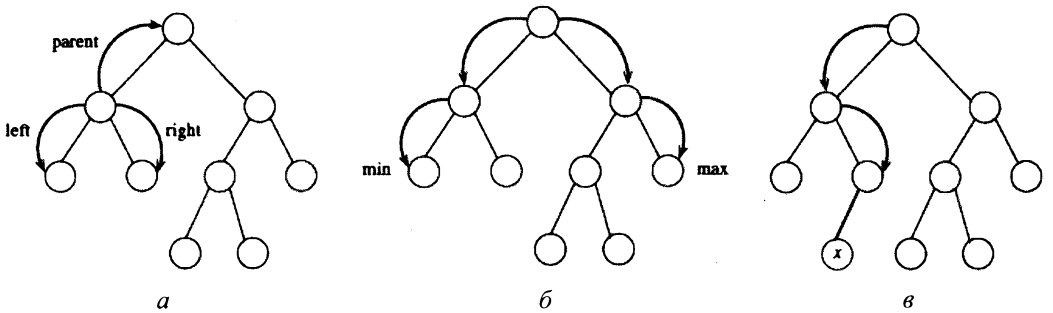



**Рис. 3.4.** Отношения в двоичном дереве: *а* — родительский и дочерние элементы; *б* — поиск в двоичном дереве наименьшего и наибольшего элементов; *в* — вставка нового элемента в правильное место

**Листинг 3.6. Объявление типа для структуры дерева**

```
typedef struct tree {
   item_type item;          /* Элемент данных */
   struct tree *parent;     /* Указатель на родительский узел */
   struct tree *left;       /* Указатель на левый дочерний узел */
   struct tree *right;      /* Указатель на правый дочерний узел */
} tree;
```

Двоичные деревья поддерживают три основные операции: поиск, обход, вставку и удаление.

## Поиск в дереве

Схема маркировки двоичного дерева поиска однозначно определяет расположение каждого ключа. Поиск начинается с корневого узла дерева. Если узел не содержит искомый ключ $x$, то в зависимости от того, меньше или больше значение искомого ключа значения ключа корневого узла, поиск продолжается в левом или правом дочернем узле соответственно. Этот алгоритм основан на том, что как левое, так и правое поддерево двоичного дерева сами являются двоичными деревьями. Реализация такого рекурсивного алгоритма поиска в двоичном дереве приведена в листинге 3.7. Время исполнения этого алгоритма равно $O(h)$, где $h$ обозначает высоту дерева.

**Листинг 3.7. Алгоритм рекурсивного поиска произвольного элемента в двоичном дереве**

```
tree *search_tree(tree *l, item_type x) {
    if (l == NULL) {
        return(NULL);
    }
```

```
    if (l->item == x) {
        return(l);
    }

    if (x < l->item) {
        return(search_tree(l->left, x));
    } else {
        return(search_tree(l->right, x));
    }
}
```

## Поиск наименьшего и наибольшего элементов дерева

Согласно определению двоичного дерева наименьший ключ должен находиться в левом поддереве корневого узла, поскольку значения всех ключей в этом дереве меньше, чем значение ключа корневого узла. Поэтому, как показано на рис. 3.4, *б*, наименьший элемент должен быть самым левым потомком корневого узла. Аналогично наибольший элемент должен быть самым правым потомком корневого узла (см. рис. 3.3, *б*).

Алгоритм поиска наименьшего элемента двоичного дерева приведен в листинге 3.8.

**Листинг 3.8. Поиск наименьшего элемента в двоичном дереве**

```
tree *find_minimum(tree *t) {
    tree *min; /* Указатель на наименьший элемент */

    if (t == NULL) {
        return(NULL);
    }

    min = t;
    while (min->left != NULL) {
        min = min->left;
    }
    return(min);
}
```

## Обход дерева

Посещение всех узлов двоичного дерева является важной составляющей многих алгоритмов. Эта операция представляет собой частный случай задачи обхода всех узлов и ребер графа, обсуждению которой посвящена *глава 7*.

Основное применение алгоритма обхода дерева — это перечисление ключей всех его узлов. Природа двоичного дерева позволяет с легкостью перечислить ключи всех его узлов в *отсортированном* порядке. В соответствии с определением двоичного дерева все ключи со значением меньшим, чем значение ключа корневого узла, должны находиться в левом поддереве корневого узла, а все ключи с большим значением — в правом. Таким образом, результатом рекурсивного посещения узлов согласно указанному принципу является *симметричный* обход (in-order traversal) двоичного дерева. Реализация соответствующего алгоритма приведена в листинге 3.9.

**Листинг 3.9. Рекурсивный алгоритм симметричного обхода двоичного дерева**

```
void traverse_tree(tree *l)
{
   if (l != NULL) {
      traverse_tree(l->left);
      process_item(l->item);
      traverse_tree(l->right);
   }
}
```

В процессе обхода алгоритм посещает каждый элемент только один раз, поэтому он исполняется за время $O(n)$, где $n$ обозначает количество элементов в дереве.

Разные типы обхода возникают вследствие изменения расположения элемента `process_item` относительно левого и правого поддеревьев. Если элемент обрабатывается первым, имеем *прямой* обход, а если последним — *обратный* обход. Для деревьев поиска эти типы обхода не имеют большого смысла, но оказываются полезными, когда дерево с корнем представляет арифметические или логические выражения.

## Вставка элементов в дерево

В двоичном дереве поиска $T$ имеется только одно место, в которое можно вставить новый элемент $x$, так что мы точно знаем, где его потом можно будет найти в случае необходимости. Для этого нужно заменить указателем на вставляемый элемент указатель `NULL`, возвращенный неуспешным запросом поиска ключа элемента $x$ в дереве.

В реализации этой операции этапы поиска и вставки узла совмещаются с помощью рекурсии. Соответствующий алгоритм приведен в листинге 3.10.

**Листинг 3.10. Вставка узла в двоичное дерево поиска**

```
void insert_tree(tree **l, item_type x, tree *parent) {
    tree *p; /* temporary pointer */

    if (*l == NULL) {
        p = malloc(sizeof(tree));
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;
        return;
    }

    if (x < (*l)->item) {
        insert_tree(&((*l)->left), x, *l);
    } else {
        insert_tree(&((*l)->right), x, *l);
    }
}
```

Процедура `insert_tree` принимает три аргумента:

- указатель `l` на указатель, связывающий поддерево с остальной частью дерева;
- вставляемый ключ `x`;
- указатель `parent` на родительский узел, содержащий `l`.

Вставляемому узлу выделяется память, и он интегрируется в структуру дерева после нахождения указателя со значением `NULL`. Обратите внимание, что во время поиска соответствующему левому или правому указателю передается *указатель*, так что операция присваивания `*l = p;` интегрирует новый узел в структуру дерева.

После выполнения поиска за время $O(h)$ выделение памяти новому узлу и интегрирование его в структуру дерева выполняется за постоянное время. В нашем случае $h$ обозначает высоту дерева поиска.

### Удаление элемента из дерева

Операция удаления элемента несколько сложнее, чем вставка, поскольку удаление узла означает, что нужно должным образом связать получившиеся поддеревья с общим деревом. Существуют три типа удаления, показанные на рис. ЦВ-3.5. Внимательно изучите этот рисунок.

Так как листовые узлы не имеют потомков, то их можно удалить, просто обнулив указатель на такой узел (см. рис. ЦВ-3.5, *б*). Удаление узла с одним потомком (см. рис. ЦВ-3.5, *в*) также не вызывает проблем. Этот потомок можно связать с родителем удаленного узла, не нарушая при этом симметричную схему размещения ключей дерева.

Но как удалить узел с двумя потомками (см. рис. ЦВ-3.5, *г*)? Нужно присвоить этому узлу ключ его непосредственного потомка в отсортированном порядке. Этот дочерний узел должен иметь наименьшее значение ключа в правом поддереве, т. е. быть самым левым узлом в правом поддереве родительского дерева `p`. Перемещение этого потомка на место удаляемого узла дает в результате двоичное дерево с должным расположением ключей, а также сводит нашу задачу к физическому удалению узла с, самое большее, одним потомком, а это задача уже была решена ранее.

Полная реализация не приводится здесь по той причине, что она выглядит несколько устрашающе, хотя ее код логически следует из приведенного описания.

Как определить сложность наихудшего случая? Для каждого удаления требуется самое большее две операции поиска, каждая из которых выполняется за время $O(h)$, где $h$ — высота дерева, плюс постоянные затраты времени на манипуляцию указателями.

## 3.4.2. Эффективность двоичных деревьев поиска

Все три словарные операции, реализованные посредством двоичных деревьев поиска, исполняются за время $O(h)$, где $h$ — высота дерева. Дерево имеет самую меньшую высоту, на которую мы можем надеяться, когда оно полностью сбалансированно, означая, что высота $h = \lceil \log n \rceil$. Это очень хорошо, но дерево должно быть идеально сбалансированным.

Наш алгоритм вставки помещает каждый новый элемент в лист дерева, в котором он должен находиться. Вследствие этого форма (и, что более важно, высота) дерева определяется порядком вставки ключей.

К сожалению, при построении дерева методом вставок могут происходить неприятные события, поскольку структура данных не контролирует порядок вставок. Например, посмотрим, что случится, если вставлять ключи в отсортированном порядке. Здесь операции `insert(a)`, `insert(b)`, `insert(c)`, `insert(d)` и т. д. дадут нам тонкое длинное дерево, в котором используются только правые узлы.

Таким образом, высота двоичных деревьев может лежать в диапазоне от $\lg n$ до $n$. Но какова средняя высота дерева? Что именно мы считаем средней высотой? Вопрос будет четко определен, если предположить равновероятными каждое из $n!$ возможных упорядочиваний вставки и взять среднее их значение. Если наше предположение окажется правильным, то нам повезло, т. к. с высокой вероятностью высота получившегося дерева окажется равной $\Theta(\log n)$. Это будет показано в *разд. 4.6*.

Приведенный пример наглядно демонстрирует мощь *рандомизации*. Часто можно разработать простые алгоритмы, с высокой вероятностью имеющие хорошую производительность. Далее мы увидим, что подобная идея лежит в основе алгоритма быстрой сортировки quicksort.

## 3.4.3. Сбалансированные деревья поиска

Произвольные деревья поиска *обычно* дают хорошие результаты. Но если нам не повезет с порядком вставок, то в наихудшем случае мы может получить дерево линейной высоты. Мы не можем прямо контролировать этот наихудший случай, поскольку должны создавать дерево в ответ на запросы, исходящие от пользователя.

Но ситуацию можно улучшить с помощью процедуры вставки/удаления, которая после каждой вставки слегка корректирует дерево, удерживая его как можно более сбалансированным, чтобы максимальная высота дерева была логарифмической. Существуют сложные структуры данных в виде *сбалансированных* двоичных деревьев поиска, которые гарантируют, что высота дерева всегда будет $O(\log n)$. Вследствие этого время исполнения любой из словарных операций (вставка, удаление, запрос) будет равняться $O(\log n)$. Реализация структур данных сбалансированных деревьев, таких как красно-черные и косые деревья, обсуждается в *разд. 15.1*.

С точки зрения разработки алгоритмов важно знать, что такие деревья существуют и что их можно использовать в качестве черного ящика, чтобы реализовать эффективный словарь. При расчете трудоемкости словарных операций для анализа алгоритма можно предположить, что сложность сбалансированного двоичного дерева в наихудшем случае будет подходящей мерой.

#### Подведение итогов

Выбор неправильной структуры данных для поставленной задачи может иметь катастрофические последствия для производительности. Определение самой лучшей структуры данных не настолько критично, поскольку возможно существование нескольких вариантов структур, имеющих примерно одинаковую производительность.

**ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ:**

## Использование сбалансированных деревьев поиска

**ЗАДАЧА**. Нужно прочитать $n$ чисел и вывести их в отсортированном порядке. Допустим, что у нас имеется сбалансированный словарь, который поддерживает операции `search`, `insert`, `delete`, `minimum`, `maximum`, `successor` и `predecessor`, время исполнения каждой из которых равно $O(\log n)$. Отсортируйте данные за время $O(n \log n)$, используя только:

1. Операции вставки и симметричного обхода.
2. Операции `minimum`, `successor` и `insert`.
3. Операции `minimum`, `insert`, `delete` и `search`.

**РЕШЕНИЕ**. Любой алгоритм сортировки посредством двоичного дерева поиска *обязан* начать с построения самого дерева. Для этого нужно инициализировать дерево (по сути, присвоив указателю $t$ значение `NULL`), а затем считывать/вставлять каждый из $n$ элементов в позицию $t$. Это занимает время $O(n \log n)$, поскольку каждая вставка занимает максимум $O(\log n)$. Довольно любопытно, что сам шаг создания структуры данных ограничивает скорость всех наших алгоритмов сортировки.

В первой задаче мы можем выполнять операции вставки и симметричного обхода. Это позволяет нам создать дерево поиска, вставив все $n$ элементов, после чего выполнить обход дерева, чтобы отсортировать элементы.

Во второй задаче, после создания дерева, мы можем использовать операции `minimum` и `maximum`. Для сортировки мы можем выполнить обход дерева, начав с наименьшего элемента и последовательно выполняя операцию поиска следующего элемента.

В третьей задаче мы не имеем операции поиска следующего элемента, но можем выполнять удаление. Для сортировки мы можем выполнить обход дерева, опять начав с наименьшего элемента и последовательно выполняя операцию удаления следующего наименьшего элемента.

Таким образом, решения этих трех задач следующие:

```
Sort1()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t)
  traverse(t)
```

```
Sort2()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t);
  y = minimum(t)
  while (y≠NULL) do
    print(y→item)
    y = successor(y,t)
```

```
Sort3()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t);
  y = minimum(t)
  while (y≠NULL) do
    print(y→item)
    delete(y,t)
    y = minimum(t)
```

Каждый из приведенных алгоритмов выполняет линейное количество операций с логарифмическим временем исполнения, что дает общее время его исполнения $O(n \log n)$. Основной подход к использованию сбалансированных двоичных деревьев заключается в том, чтобы рассматривать их как «черные ящики». ■

# 3.5. Очереди с приоритетами

Многие алгоритмы обрабатывают элементы в определенном порядке. Допустим, например, что нужно запланировать выполнение работ согласно их относительной важности. Для этого работы нужно сначала отсортировать по важности, после чего обработать их в этом отсортированном порядке.

Абстрактный тип данных, называющийся *очередью с приоритетами*, предоставляет разработчику больше гибкости, чем обычная сортировка, т. к. позволяет вводить новые элементы в систему в произвольном месте. Вставить новый элемент в нужное место в очереди с приоритетами может оказаться намного эффективнее, чем сортировать заново все элементы при каждой вставке.

Базовая очередь с приоритетами поддерживает три основные операции:

- `insert(Q,x)` — вставляет в очередь с приоритетами `Q` элемент `x`;
- `find-minimum(Q)` и `find-maximum(Q)` — возвращает указатель на элемент с наименьшим (наибольшим) значением ключа среди всех ключей в очереди с приоритетами `Q`;
- `delete-minimum(Q)` и `delete-maximum(Q)` — удаляет из очереди с приоритетами `Q` элемент с наименьшим (наибольшим) значением ключа.

Очереди с приоритетами часто неформально моделируют природные процессы. Например, люди с неустроенной личной жизнью мысленно (или открыто) формируют очередь с приоритетами из возможных кандидатов на роль постоянного близкого человека. Впечатления о новом знакомстве отображаются непосредственно на шкале привлекательности. Привлекательность здесь играет роль *ключа* для создания нового элемента в очереди с приоритетами. Выбор постоянного близкого человека — это процесс, включающий в себя извлечение наиболее привлекательного элемента из очереди (выполняется операция `find-maximum`), свидание с ним для уточнения степени привлекательности и вставку обратно в очередь с приоритетами, возможно, в другое место с новым значением привлекательности.

#### *Подведение итогов*

Словари и очереди с приоритетами позволяют создавать алгоритмы с аккуратной структурой и хорошей производительностью.

**Остановка для размышлений:**

### Построение базовых очередей с приоритетами

**Задача.** Определить временную сложность в наихудшем случае трех основных операций очереди с приоритетами (вставка элемента, поиск наименьшего элемента и поиск наибольшего элемента) для следующих базовых структур данных:

- неотсортированного массива;
- отсортированного массива;
- сбалансированного двоичного дерева.

**РЕШЕНИЕ**. В реализации этих трех операций присутствуют некоторые тонкости, даже при использовании такой простой структуры данных, как неотсортированный массив. В словаре на основе неотсортированного массива (см. *разд. 3.3*) операции вставки и удаления выполняются за постоянное время, а операции поиска произвольного и наименьшего элементов — за линейное. Операцию удаления наименьшего элемента (`delete-minimum`), выполняемую за линейное время, можно составить из последовательности операций `find-minimum` и `delete`.

В отсортированном массиве операции вставки и удаления можно выполнить за линейное время, а найти наименьший элемент — за постоянное. Но удаления из очереди с приоритетами затрагивают только наименьший элемент. После сортировки массива в обратном порядке (наибольший элемент вверху) наименьший элемент всегда будет самым последним элементом массива. Чтобы удалить последний элемент, не нужно перемещать никакие элементы, а только уменьшить на единицу количество оставшихся элементов $n$, вследствие чего операция удаления наименьшего элемента может выполняться за постоянное время.

Все это правильно, но, как можно видеть из табл. 3.3, операцию поиска наименьшего значения, выполняемую за постоянное время, можно реализовать для каждой структуры данных.

**Таблица 3.3.** Операции поиска наименьшего значения, выполняемые за постоянное время

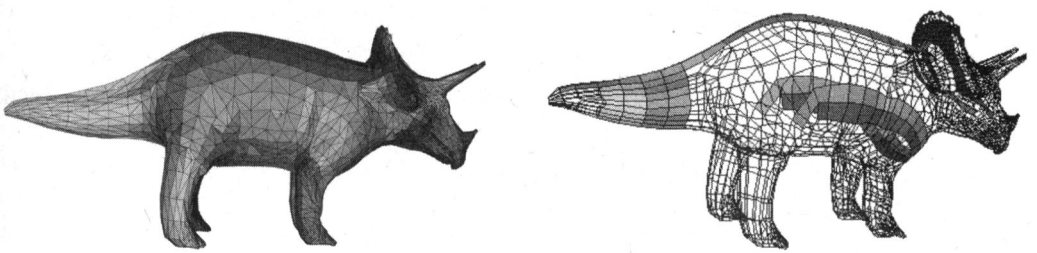
| Операция | Неотсортированный массив | Отсортированный массив | Сбалансированное дерево |
|---|---|---|---|
| `insert(Q,x)` | $O(1)$ | $O(n)$ | $O(\log n)$ |
| `find-minimum(Q)` | $O(1)$ | $O(1)$ | $O(1)$ |
| `delete-minimum(Q)` | $O(n)$ | $O(1)$ | $O(\log n)$ |

Вся хитрость заключается в использовании дополнительной переменной для хранения указателя на наименьший элемент в каждой из указанных структур, что позволяет просто возвратить это значение в любое время при получении запроса на поиск такого элемента. Обновление этого указателя при каждой вставке не составляет труда — он обновляется тогда и только тогда, когда вставленный элемент меньше, чем текущий наименьший элемент. Но что будет в случае удаления наименьшего элемента? Мы можем удалить текущий наименьший элемент, после чего найти новый наименьший элемент и зафиксировать его в этом качестве до тех пор, пока он тоже не будет удален. Операция поиска нового наименьшего элемента занимает линейное время для неотсортированных массивов и логарифмическое время для деревьев, и поэтому затраты на ее выполнение можно включить в затраты на выполнение каждого удаления.

Очереди с приоритетами являются очень полезными структурами данных. Особенно изящная реализация очереди с приоритетами — пирамида — рассматривается в контексте задачи сортировки в *разд. 4.3*. Кроме этого, в *разд. 15.2* дается полный набор реализаций очереди с приоритетами. ■

# 3.6. История из жизни. Триангуляция

Применяемые в компьютерной графике геометрические модели обычно разбивают на множество небольших треугольников, как показано на рис. 3.6, *а*.



*а* *б*

**Рис. 3.6.** Триангуляционная модель динозавра (*а*); использование нескольких полос треугольников (*б*)

Для прорисовывания и закрашивания треугольников применяются высокопроизводительные движки визуализации (rendering engine), работающие на специализированном аппаратном обеспечении. Скорость работы этого аппаратного обеспечения для визуализиции такая высокая, что единственным узким местом в системе являются затраты на ввод в него триангулированной структуры.

Хотя каждый треугольник можно описать, указав три его вершины, существует и альтернативное представление, которое оказывается более эффективным. Вместо того чтобы определять каждый треугольник по отдельности, мы соединяем их в *полосы* смежных треугольников (рис. 3.6, *б*) и обрабатываем их, перемещаясь вдоль этой полосы. Поскольку в таком случае каждый треугольник имеет две общие вершины со своим соседом, то мы экономим на расходах по передаче данных о двух вершинах и другой сопутствующей информации. Для однозначного описания треугольников в рендерере треугольной сетки OpenGL принимается соглашение, что все повороты чередуются влево-вправо.

Полоса на рис. 3.7, *а* описывает пять треугольников сетки с последовательностью вершин [1, 2, 3, 4, 5, 7, 6] и подразумеваемым упорядочиванием слева направо. А полоса на рис. 3.7, *б* описывает все семь треугольников с указанными поворотами: [1, 2, 3, L-4, R-5, L-7, R-6, R-1, R-3].

Задача состояла в определении минимального количества полос, которые совместно покрывают все треугольники в сетке, не перекрывая друг друга. Это можно рассматривать как задачу графов. В таком графе каждый треугольник сетки представляется вершиной, а смежные треугольники — ребром между соответствующими вершинами. Такое представление в виде *двойственного графа* содержит всю информацию, необходимую для разбиения треугольной сетки на полосы (см. *разд. 18.12*).

После получения двойственного графа можно было приступать к работе. Мы хотели разбить множество вершин двойственного графа на минимально возможное количество путей (или полос). Если бы мы смогли объединить их в один путь, то это бы означало, что мы нашли *гамильтонов путь*, т. е. путь, содержащий каждую вершину графа

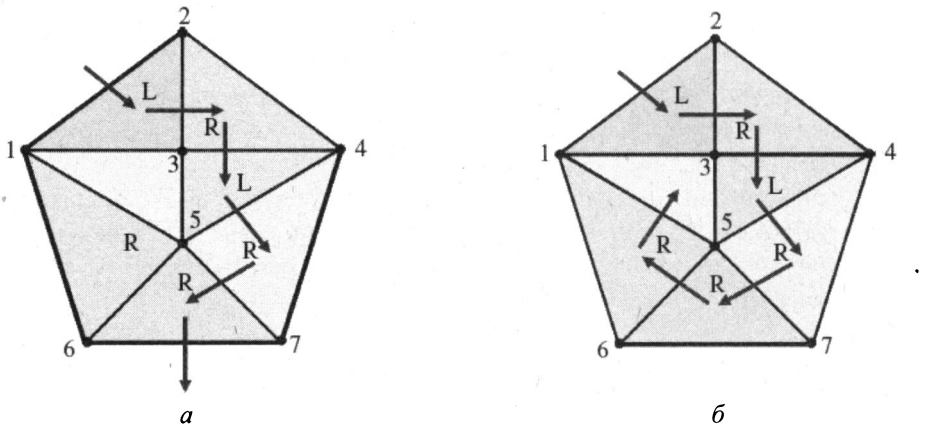



**Рис. 3.7.** Извлечение полосы треугольников из треугольной сетки: полоса с частичным покрытием с использованием чередующихся левых и правых поворотов (*а*) и полоса с полным покрытием с использованием гибкости произвольных поворотов (*б*)

ровно один раз. Поскольку поиск гамильтонова пути является NP-полной задачей (см. *разд. 19.5*), мы знали, что нам не стоит искать алгоритм точного решения, а лучше сконцентрироваться на эвристическом алгоритме приблизительного решения.

Самый простой подход для полосового покрытия — начать работу с произвольного треугольника и двигаться направо, пока не будет достигнута граница объекта или уже посещенный треугольник. Достоинством этого эвристического алгоритма являются его быстрота и простота, но при этом нет оснований ожидать, что ему удастся найти наименьший набор ориентированных слева направо полос для заданной триангуляции.

А вот использование «жадного» эвристического алгоритма должно сделать получение сравнительно небольшого количества полос более вероятным. «Жадные» алгоритмы выбирают оптимальные возможные решения на каждом этапе в надежде, что и конечное решение также будет оптимальным. При выполнении триангуляции «жадный» алгоритм сначала определяет начальный треугольник самой длинной полосы слева направо и отделяет ее.

Но сам факт использования «жадного» алгоритма не гарантирует наилучшее возможное общее решение, поскольку первая отделенная полоса может разбить много потенциальных полос, которые можно было бы использовать в дальнейшем. Тем не менее хорошим практическим правилом для получения оптимального решения все же является использование «жадного» алгоритма. Отделение самой длинной полосы оставляет самое меньшее количество треугольников для создания последующих полос, поэтому «жадный» эвристический алгоритм должен быть более производительным, чем простой эвристический алгоритм для выбора любых элементов.

Но сколько времени понадобится этому алгоритму, чтобы найти следующую самую длинную полосу треугольников? Пусть $k$ — длина прохода, начинающегося в средней вершине. Используя самую простую реализацию алгоритма, можно выполнять проход от каждой из $n$ вершин, что позволит найти самую длинную оставшуюся полосу за время $O(kn)$. Повторение такого прохода для каждой из почти $n/k$ извлекаемых полос

дает нам время исполнения $O(n^2)$, что неприемлемо медленно даже для небольшой модели, состоящей из 20 тысяч треугольников.

Как можно улучшить это время? Интуитивно кажется, что неэкономично повторять проход от каждого треугольника после удаления лишь одной полосы. Можно просто хранить информацию о длине всех возможных будущих полос в какой-либо структуре данных. Но при удалении каждой полосы необходимо обновить информацию о длине всех других затронутых таким удалением полос. Эти полосы будут укорочены, поскольку проходят через треугольник, которого больше нет. Такая структура данных будет иметь характеристики двух структур:

- *Очередь с приоритетами.*

  Так как мы многократно повторяем процесс определения самой длинной оставшейся полосы, то нам нужна очередь с приоритетами для хранения полос, упорядоченных по длине. Следующая удаляемая полоса всегда находится в начале очереди. Наша очередь с приоритетами должна была разрешать понижение приоритета произвольных элементов в очереди при каждом обновлении информации о длине полос, чтобы мы знали, какие треугольники были удалены. Поскольку длина всех полос ограничивалась весьма небольшим целым числом (аппаратные возможности не позволяли иметь в полосе более 256 вершин), мы использовали ограниченную по высоте очередь с приоритетами (массив корзин, показанный на рис. 3.8 и рассматриваемый в *разд. 15.2*). Обычная куча также была бы вполне приемлемой.

  Для обновления элемента очереди, связанного с каждым треугольником, требовалось быстро находить его. Это означало, что нам нужен словарь.

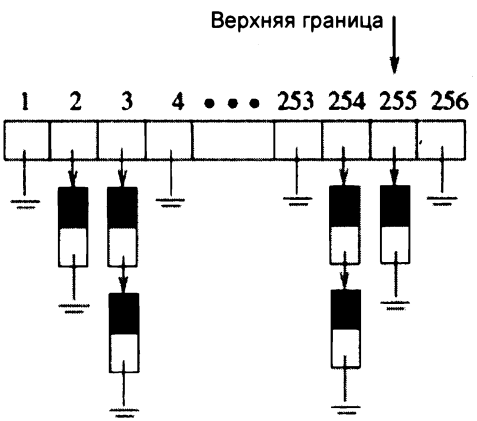



**Рис. 3.8.** Ограниченная по высоте очередь с приоритетами для полос треугольников

- *Словарь.*

  Для каждого треугольника в сетке нам надо было знать его расположение в очереди. Это означало, что в словаре следовало иметь указатель на каждый треугольник. Объединив этот словарь с очередью с приоритетами, мы создали структуру данных, способную поддерживать широкий диапазон операций.

Хотя возникали и другие трудности — например, необходимость найти способ быстрого вычисления новой длины у полос, затронутых удалением полосы, главный метод повышения производительности состоял в применении очереди с приоритетами. Использование этой структуры данных улучшило время исполнения на несколько порядков.

Из табл. 3.4 видно, насколько производительность «жадного» алгоритма выше, чем у простого.

**Таблица 3.4.** Сравнение производительности «жадного» и простого алгоритмов для нескольких типов сетки треугольников. Затраты измеряются количеством полос. Время исполнения обычно масштабируется по количеству треугольников, за исключением в высшей степени симметричного тора с очень длинными полосами

| Название модели | Количество треугольников | Затраты простого алгоритма | Затраты «жадного» алгоритма | Время «жадного» алгоритма |
|---|---|---|---|---|
| Ныряльщик | 3798 | 8460 | 4650 | 6,4 с |
| Головы | 4157 | 10 588 | 4749 | 9,9 с |
| Каркас | 5602 | 9274 | 7210 | 9,7 с |
| Барт Симпсон | 9654 | 24 934 | 11 676 | 20,5 с |
| Космический корабль Enterprise | 12 710 | 29 016 | 13 738 | 26,2 с |
| Тор | 20 000 | 40 000 | 20 200 | 272,7 с |
| Челюсть | 75 842 | 104 203 | 95 020 | 136,2 с |

Во всех случаях «жадный» алгоритм выдавал набор полос меньшей стоимости в смысле общего количества вершин в полосах. Экономия составляла от 10 до 50%, что было просто замечательно, поскольку максимально возможное улучшение (уменьшение количества вершин каждого треугольника с трех до одной) позволяет сэкономить не более 66,6%.

Результатом реализации «жадного» алгоритма со структурой данных в виде очереди с приоритетами было время исполнения программы $O(n \cdot k)$, где $n$ — количество треугольников, $k$ — длина средней полосы. Вследствие этого обработка тора, состоящего из небольшого количества очень длинных полос, занимала больше времени, чем обработка челюсти, хотя количество треугольников в последней фигуре было втрое больше.

Из этой истории можно извлечь несколько уроков. Во-первых, при работе с достаточно большим набором данных только алгоритмы с линейным или почти линейным временем исполнения могут быть достаточно быстрыми. Во-вторых, выбор правильной структуры данных часто является ключевым фактором для уменьшения временной сложности алгоритма. И в-третьих, применение «жадного» эвристического алгоритма может значительно улучшить производительность по сравнению с применением простого алгоритма. Насколько большим будет это улучшение, можно определить только экспериментальным путем.

# 3.7. Хеширование

Хеш-таблицы предоставляют *очень* практичный способ реализации словарей. В них используется то обстоятельство, что поиск элемента массива по индексу выполняется за постоянное время. *Хеш-функция* соотносит набор ключей с целочисленными значениями. Мы будем использовать значение нашей хеш-функции в качестве индекса массива и записывать элемент в этой позиции.

Сначала хеш-функция соотносит каждый ключ (в нашем случае строку $S$) с большим целым числом. Пусть значение $\alpha$ представляет размер алфавита, используемого для создания строки $S$. Пусть `char(c)` будет функцией, которая однозначно отображает каждый символ алфавита в целое число в диапазоне от 0 до $\alpha - 1$. Функция

$$H(S) = \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

однозначно отображает каждую строку в (большое) целое число, рассматривая символы строки как «цифры» системы счисления с основанием $\alpha$.

В результате получатся уникальные идентификационные числа, но они будут настолько большими, что очень быстро превысят количество требуемых ячеек в нашей хеш-таблице (обозначаемое $m$). Это значение необходимо уменьшить до целого числа в диапазоне от 0 до $m - 1$, для чего выполняется операция получения остатка от деления $H'(S) = H(S) \bmod m$. Здесь применяется тот же самый принцип, что и в колесе рулетки. Шарик проходит долгий путь, обходя $\lfloor H(S)/m \rfloor$ раз колесо окружностью $m$, пока не остановится в произвольной ячейке, размер которой составляет незначительную часть пройденного пути. Если выбрать размер таблицы с достаточной тщательностью (в идеале число $m$ должно быть большим простым числом, не слишком близким к $2^i - 1$), то распределение полученных хеш-значений будет достаточно равномерным.

## 3.7.1. Коллизии

Независимо от того, насколько хороша наша хеш-функция, она будет, по крайней мере, время от времени отображать два разных ключа в одно хеш-значение, и нужно быть готовым к подобной ситуации. Хеш-таблицу можно реализовать двумя разными способами:

- *С использованием цепочек.* Цепочка представляет массив из $m$ связных списков (корзин), как показано на рис. 3.9.

  Список с порядковым номером $i$ содержит все элементы, хешированные в одно и то же значение $i$. Операции поиска, вставки и удаления сводятся к соответствующим операциям в связных списках. Если $n$ ключей распределены в таблице равномерно, то каждый список будет содержать приблизительно $n/m$ элементов, делая их размер постоянным при $m \approx n$.

  Метод цепочек — самый естественный, но он требует значительных объемов памяти для хранения указателей. Эту память можно было использовать, чтобы сделать таблицу больше, что понижает вероятность коллизий. Впрочем, хеш-таблицы с наи-

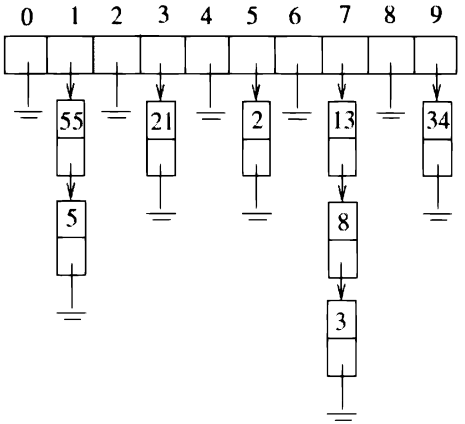



**Рис. 3.9.** Разрешение коллизий при помощи цепочек после хеширования первых восьми чисел Фибоначчи в возрастающем порядке посредством функции хеширования $H(x) = (2x + 1) \bmod 10$. На рисунке вставки делаются в начале каждого списка

большей производительностью реализуются при помощи альтернативного метода, называющегося *открытой адресацией* (см. далее).

- При использовании метода *открытой адресации* хеш-таблица реализуется в виде массива ключей (а не корзин), каждый из которых инициализирован нулевым значением, как показано на рис. ЦВ-3.10.

При каждой вставке ключа выполняется проверка, свободна ли требуемая ячейка. Если свободна, то вставка выполняется. В противном случае нам нужно найти другое место, куда вставить этот ключ. Самый простой подход к определению альтернативного места для вставки называется *последовательным исследованием* (sequential probing). Этот метод заключается в том, что последовательно исследуются следующие ячейки таблицы, пока не будет найдена свободная, в которую и выполняется вставка. При условии, что таблица не слишком заполнена, последовательность смежных занятых ячеек должна быть достаточно небольшой, поэтому свободная ячейка *должна* находиться лишь в нескольких позициях от требуемой.

Теперь для поиска определенного ключа нужно взять соответствующее хеш-значение и проверить, является ли его значение тем, какое нам нужно. Если да, то поиск заканчивается. В противном случае поиск продолжается в последовательности смежных ячеек, пока не будет найдено требуемое значение. Удаление из хеш-таблицы с открытой адресацией — довольно сложная процедура, поскольку удаление одного элемента может разорвать цепочку вставок, вследствие чего некоторые элементы последовательности больше не будут доступны. Не остается иного выхода, кроме как выполнить повторную вставку всех элементов, следующих за удаленным.

В обоих методах разрешения коллизий требуется время $O(m)$ для инициализации нулями хеш-таблицы из $m$ элементов до того, как можно будет выполнять первую вставку.

При использовании метода цепочек для разрешения коллизий в хеш-таблице из $m$ элементов словарные операции для $n$ элементов можно реализовать со следующими показателями времени исполнения в ожидаемом и наихудшем случаях (табл. 3.5).

**Таблица 3.5.** Словарные операции с показателями времени исполнения в ожидаемом и наихудшем случаях

| Операция | Ожидаемое время исполнения | Время исполнения в наихудшем случае |
|---|---|---|
| `search(L,k)` | $O(n/m)$ | $O(n)$ |
| `insert(L,x)` | $O(1)$ | $O(1)$ |
| `delete(L,x)` | $O(1)$ | $O(1)$ |
| `successor(L,x)` | $O(n+m)$ | $O(n+m)$ |
| `predecessor(L,x)` | $O(n+m)$ | $O(n+m)$ |
| `minimum(L)` | $O(n+m)$ | $O(n+m)$ |
| `maximum(L)` | $O(n+m)$ | $O(n+m)$ |

Обход всех элементов в таблице цепочек занимает время $O(n+m)$, поскольку нам нужно просмотреть все $m$ корзин в процессе поиска элементов, даже если в действительности таблица содержит небольшое количество вставленных элементов. Для таблицы с открытой адресацией это время равно $O(m)$, т. к. значение $n$ может быть равным самое большее значению $m$.

С практической точки зрения, хеш-таблица часто является самым лучшим типом структуры данных для реализации словаря. Но область применения хеширования гораздо шире, чем просто создание словарей, в чем мы вскоре убедимся.

## 3.7.2. Выявление дубликатов с помощью хеширования

Основополагающей идеей хеширования является представление большого объекта (будь то ключ, строка или подстрока) посредством одного числа. В результате мы получим представление большого объекта значением, которым можно манипулировать за постоянное время, при этом вероятность представления двух разных больших объектов одним и тем же числовым значением должна быть сравнительно невысокой.

Кроме ускорения поиска хеширование находит многие другие разнообразные хитроумные применения. Я однажды слышал, как Уди Манбер (Udi Manber), в одно время ответственный за все поисковые продукты в Yahoo, рассказывал об алгоритмах, используемых в промышленности. По его словам, наиболее важными алгоритмами были хеширование, хеширование и снова хеширование.

Рассмотрим следующие задачи, имеющие красивые решения посредством хеширования.

- *Является ли тот или иной документ уникальным в большом собрании документов?*

  Поисковый механизм обрабатывает следующую веб-страницу. Как он может определить, содержит ли она новую информацию или же просто дублирует какую-либо другую страницу в Интернете?

  Для большого собрания документов будет весьма неэффективно явно сравнивать новый документ $D$ со всеми $n$ предыдущими документами. Но мы можем преобра-

зовать документ $D$ в целое число с помощью хеш-функции и сравнить $H(D)$ с хеш-кодами документов, уже имеющихся в базе данных. Только в случае коллизии хеш-значений документ $D$ *может* быть возможным дубликатом. Так как вероятность ложных коллизий низка, то мы без больших затрат можем явно сравнить те несколько документов с одинаковыми хеш-кодами.

- *Не является ли тот или иной документ плагиатом?*

  Ленивый студент копирует часть документа из Интернета в свою курсовую работу, слегка изменив его в некоторых местах. «Интернет большой, — ухмыляется он. — Как сможет кто-либо найти в нем страницу, из которой я взял этот материал?»

  Это гораздо более трудная задача, чем предыдущая. Если в документе добавлен, удален или изменен всего лишь один символ, то его хеш-код будет совершенно другим. Таким образом, подход с использованием хеш-кода, применяемый для решения предыдущей задачи, не годится для решения этой более общей задачи.

  Но мы можем создать хеш-таблицу всех перекрывающихся окон (подстрок) длиной $w$ символов для всех документов в базе данных. Любое совпадение хеш-кодов означает, что оба документа, вероятно, содержат одинаковую подстроку длиной $w$ символов, что можно исследовать более подробно. Значение $w$ нужно выбрать достаточно длинным, чтобы минимизировать вероятность случайного совпадения хеш-кодов.

  Самым большим недостатком этой схемы является то обстоятельство, что объем хеш-таблицы становится таким же большим, как и объем самого собрания документов. Эта задача решается при помощи minwise-хеширования (см. *разд. 6.6*), когда оставляется небольшое, но удачно определенное подмножество этих хеш-кодов.

- *Как убедиться в том, что файл не был изменен?*

  На закрытых торгах каждая сторона подает свое предложение цены, которое не известно никакой другой стороне. В назначенное время цены оглашаются и сторона, предложившая наивысшую цену, объявляется выигравшей торги. Если какой-либо недобросовестный участник торгов знает предложения других сторон, то он может предложить цену, немного превышающую максимальную цену, предложенную оппонентами, и выиграть торги. Одним из способов получить информацию о предложениях других сторон будет взлом компьютера, на котором хранится эта информация.

  Для предотвращения такого развития событий можно потребовать предоставления хеш-кода предложения до даты открытия предложений с тем, чтобы само предложение было представлено только после этой даты. После определения победителя его предложение хешируется и полученное хеш-значение сравнивается с хеш-значением, предоставленным ранее. Такие методы *криптографического хеширования* позволяют удостовериться, что предоставленный сегодня файл такой же, как и первоначальный, поскольку любые изменения в файле отразятся в измененном хеш-коде.

Хотя наихудшие случаи любого алгоритма с хешированием способны привести в смятение, при правильно подобранной хеш-функции мы можем с уверенностью ожидать

хорошиех результатов. Хеширование является основополагающей идеей рандомизированных алгоритмов, позволяющей получить линейное ожидаемое время исполнения алгоритма для задач с временной сложностью $\Theta(n \log n)$ или даже $\Theta(n^2)$ в наихудшем случае.

## 3.7.3. Прочие приемы хеширования

Кроме создания таблиц хеширования функции хеширования оказываются полезными для многих других задач. Фундаментальная идея заключается в сопоставлении типа «многие-к-одному», где часть *многие* управляется таким образом, чтобы вероятность наличия *слишком многих* была очень низкой.

## 3.7.4. Каноникализация

Рассмотрим словесную игру, предлагающую найти все слова из словаря, которые можно создать, переставляя буквы множества $S$. Например, из четырех букв множества $S = (a, e, k, l)$ можно создать три слова: *kale*, *lake* и *leak*.

Подумайте, как можно написать программу для нахождения слов из множества $S$, совпадающих со словами в словаре $D$ из $n$ слов. Наверное, наиболее прямолинейным подходом будет проверка каждого слова $d \in D$ на наличие всех его букв в множестве $S$. Такая проверка занимает линейное время относительно $n$ для каждого $S$, а написание соответствующей программы сопряжено с определенными сложностями.

Но что, если вместо такой проверки мы создадим хеш-строку каждого слова в $D$, упорядочив буквы слов. Тогда *kale* (а также и *lake*, и *leak*) становится *aekl*. В результате создания хеш-таблицы, содержащей в качестве ключей упорядоченные строки, все слова с одинаковым распределением букв хешируются в одной и той же корзине. Созданную таким образом хеш-таблицу можно использовать и для других наборов запроса $S$. Время исполнения каждого запроса будет пропорционально количеству совпадающих слов в множестве $D$, которое намного меньшее чем $n$.

Какой набор из $k$ букв можно использовать для создания наибольшего количества слов в словаре? Эта задача выглядит намного труднее, поскольку имеется $\alpha^k$ возможных наборов букв, где $\alpha$ обозначает количество букв в алфавите. Но заметьте, что ответ — это просто хеш-код с наибольшим количеством коллизий. Задача быстро и легко решается проходом по массиву упорядоченных хеш-кодов (или проходом по каждой корзине хеш-таблицы с цепочками).

Это хороший пример мощи *каноникализации* — сведения сложных объектов к стандартной (т. е. «канонической») форме. Преобразования строк, наподобие сведения букв к нижнему регистру или выделения корня (удаления из слова суффиксов), повышает частоту совпадений, поскольку происходит коллизия нескольких строк по одинаковому коду. Для имен используется схема каноникализации Soundex. В этой схеме разные возможные варианты написания фамилии «Skiena» — например: «Skina», «Skinnia», и «Schiena» — получают одинаковый код Soundex: S25. Схема каноникализации Soundex рассматривается более подробно в *разд. 21.4*.

В хеш-таблицах коллизии очень нежелательны, но для задач сопоставления с образом наподобие рассмотренной, коллизии как раз то, что требуется.

### 3.7.5. Уплотнение

Предположим, что нам нужно отсортировать все $n$ книг в библиотеке, но не по названиям, а по тексту. Например, при такой сортировке текст «It was a dark and stormy night...» книги [BL30] будет предшествовать тексту «What is an algorithm?...» настоящей книги. Предполагая, что средняя книга содержит $m \approx 100\ 000$ слов, такая сортировка выглядит затратной и медленной работой, поскольку каждый раз сравниваются две книги.

Но предположим, что мы представим каждую книгу ее первыми, скажем, 100 символами, и сортировать будем уже эти строки. В случае дубликатов префиксов нескольких изданий одной книги или, возможно, плагиата у нас будут коллизии, но в весьма редких случаях. После упорядочивания префиксов такие коллизии можно разрешить, сравнив полные тексты. В *разд. 17.1* рассматривается самая быстрая в мире программа сортировки на основе этого принципа.

Хеширование для уплотнения данных, когда большой объект представляется хеш-кодом небольшого размера, пример которого мы только что рассмотрели, также называется *методом отпечатков пальцев*. Работать с небольшими объектами легче, чем с большими, а хеш-код обычно сохраняет особые характеристики каждого объекта. В данном случае функция хеширования очень простая (просто берется префикс), но она создана для конкретной цели — не использовать таблицу хеширования. Более сложные функции хеширования могут понизить вероятность коллизий даже между незначительно отличающимися объектами до исчезающе малого значения.

## 3.8. Специализированные структуры данных

Все рассматриваемые до этого времени основные структуры данных представляют обобщенные наборы элементов, облегчающие доступ к данным. Эти структуры данных хорошо известны большинству программистов. Менее известны структуры данных для представления более специализированных типов объектов, таких как точки в пространстве, строки и графы.

Принципы создания этих структур данных те же, что и для основных типов структур: имеется набор основных многократно исполняемых операций, и нам нужна структура данных, позволяющая очень эффективное исполнение этих операций. Такие специализированные структуры данных весьма важны для создания высокопроизводительных алгоритмов работы с графами и геометрическими объектами, поэтому нужно знать об их существовании. Далее приводится краткое описание нескольких таких специализированных структур данных.

- *Строки.*

  Строки обычно представляются в виде массивов символов, возможно, с использованием специального символа для обозначения конца строки. Примером таких структур данных является суффиксное дерево/массив, в котором выполняется предварительная обработка строк для ускорения операции поиска подстрок. Подробно об этом рассказано в *разд. 15.3*.

- *Геометрические структуры данных.*

  Геометрические данные обычно представляют собой коллекцию точек и областей данных. Плоскость можно представить в виде многоугольника с границей, состоящей из замкнутой цепочки отрезков. Многоугольник $P$ можно представить с помощью массива точек $(v_1, ..., v_n, v_1)$, где $(v_i, v_{i+1})$ является сегментом границы многоугольника $P$. В пространственных структурах данных, таких как *kd*-деревья, точки и области организованы по их геометрическому расположению для ускорения операций поиска. Подробно об этом рассказано в *разд. 15.6*.

- *Графы.*

  Графы обычно представляются посредством матриц или списков смежных вершин графа. Выбор конкретного типа представления может существенно повлиять на структуру конечных алгоритмов для работы с графом, как показано в *главе 7* и в *разд. 15.4*.

- *Множества.*

  Подмножества элементов — для обеспечения быстроты поиска — обычно представляются посредством словаря. Кроме того, могут использоваться *двоичные векторы*, которые представляют собой булевы массивы, в которых установленный (имеющий значение 1) $i$-й бит означает, что $i$ является членом подмножества. Структуры данных для манипулирования множествами представлены в *разд. 15.5*.

## 3.9. История из жизни. Геном человека

В геноме человека закодирована вся необходимая для создания человека информация. Секвенирование генома *человека* уже оказало огромное влияние на развитие современной медицины и молекулярной биологии. Алгоритмисты наподобие меня также заинтересовались проектом генома человека, и на то были свои причины.

- Последовательности ДНК можно точно представить в виде строк алфавита из четырех символов: A, C, T и G. Нужды биологов возродили интерес к старым алгоритмическим задачам, таким как поиск подстрок (см. *разд. 21.3*), а также создали новые задачи, такие как поиск самой короткой общей подстроки (см. *разд. 21.9*).
- Последовательности ДНК являются *очень* длинными строками. Длина генома человека составляет приблизительно три миллиарда базовых пар (или символов). Такой большой размер входного экземпляра задачи означает, что в биологических приложениях обычно совершенно необходим асимптотический анализ сложности алгоритма.
- В геномику вливается достаточно большой объем денег, что вызывает у специалистов в компьютерной области желание получить свою долю.

Лично меня в вычислительной биологии заинтересовал метод, предложенный для секвенирования ДНК и названный *секвенированием путем гибридизации* (sequencing by hybridization, SBH). Для его реализации набор проб (коротких нуклеотидных последовательностей) организуется в массив, представляющий собой *секвенирующий чип*. Каждая из этих проб определяет наличие строки пробы в виде подстроки в целевой

ДНК. Теперь можно выполнить секвенирование целевой ДНК на основе ограничений, определяющих, какие строки являются подстроками целевой ДНК.

По имеющемуся набору всех подстрок длиной $k$ строки $S$ мы должны были идентифицировать все строки длиной $2k$ как возможные подстроки $S$. Предположим, нам известно, что строки *AC*, *CA* и *CC* являются двухсимвольными подстроками строки $S$. Возможно, что строка *ACCA* является подстрокой строки $S$, поскольку средняя подстрока является одним из вариантов. Но строка *CAAC* не может быть подстрокой $S$ из-за того, что ее часть *AA* не является подстрокой $S$. Так как строка $S$ могла быть очень длинной, нам нужно было найти быстрый алгоритм для построения всех допустимых строк длиной $2k$.

Самым простым решением была бы конкатенация всех $O(n^2)$ пар строк длиной $k$ с последующей проверкой, что все $k-1$ строк длиной $k$, переходящих границу конкатенации, действительно являются подстроками (рис. 3.11).

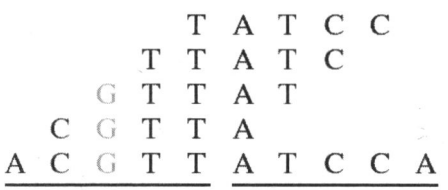



**Рис. 3.11.** Конкатенация двух строк может входить в строку $S$, только если в нее входят все объединяемые строки

Например, для подстрок *AC*, *CA* и *CC* возможны девять вариантов объединения: *ACAC*, *ACCA*, *ACCC*, *CAAC*, *CACA*, *CACC*, *CCAC*, *CCCA* и *CCCC*. Из них можно исключить только последовательность *CAAC*, поскольку *AA* не является первоначальной подстрокой строки $S$.

Нам нужно было найти быстрый способ проверить, что $k-1$ подстрок, переходящих границу конкатенации, были членами нашего словаря разрешенных строк длиной $k$. Время выполнения такого поиска зависит от структуры используемого словаря. Двоичное дерево поиска позволило бы найти правильную строку за $O(\log n)$ сравнений, состоящих из проверок, какая из двух строк длиной $k$ символов встречается первой (по алфавиту). Общее время исполнения с применением такого двоичного дерева было бы равно $O(k \log n)$.

Все эти рассуждения представлялись вполне приемлемыми, и мой аспирант Димитрис Маргаритис (Dimitris Margaritis) создал алгоритм поиска с использованием двоичного дерева. Алгоритм выглядел прекрасно, пока мы не испытали его в действии.

— Я выполнил программу на самом мощном компьютере из имеющихся в нашем распоряжении, но она слишком медленная, — пожаловался мне Димитрис. — Работает бесконечно долго на строках длиной всего лишь в 2000 символов. Нам никогда не дойти до $n = 50\,000$.

Мы исследовали нашу программу на профилировщике и обнаружили, что почти все время уходило на поиск в структуре данных. Это нас не удивило, поскольку эта операция выполнялась $k-1$ раз для каждой из $O(n^2)$ возможных последовательностей. Нам

была нужна более быстрая структура данных, т. к. поиск был самой внутренней операцией из множества вложенных циклов.

— А если попробовать хеш-таблицу? — предложил я. — Хеширование строки длиной в $k$ символов и поиск ее в нашей таблице должны занять время $O(k)$. Это сократит время исполнения до $O(\log n)$.

Димитрис опять принялся за работу и реализовал хеш-таблицу для нашего словаря. Как и в первый раз, программа выглядела прекрасно, пока мы не испытали ее в действии.

— Программа все еще работает слишком медленно, — опять пожаловался Димитрис. — Конечно же, сейчас на строках длиной в 2000 символов она выполняется раз в десять быстрее, так что мы можем дойти до строк длиной около 4000 символов. Но мы все равно никогда не сможем дойти до 50 000.

— Нам следовало ожидать этого, — размышлял я. — В конце концов, $\lg_2(2000) \approx 11$. Нам нужна более быстрая структура данных для поиска строк в словаре.

— Но что может быть быстрее, чем хеш-таблица? — возразил Димитрис. — Чтобы найти строку длиной в $k$ символов, нам нужно считать все эти $k$ символов. Наша хеш-таблица уже дает нам время поиска $O(k)$.

— Конечно же, чтобы проверить первую подстроку, нужно выполнить $k$ сравнений. Но, возможно, мы можем улучшить производительность на последующих проверках. Вспомни, как получаются наши запросы. Для конкатенации подстрок *ABCD* и *EFGH* мы сначала ищем в словаре части *BCDE*, а потом *CDEF*. Эти две подстроки отличаются всего лишь одним символом. Мы должны использовать это обстоятельство, чтобы каждая последующая проверка выполнялась за постоянное время.

— С хеш-таблицей этого сделать нельзя, — заметил Димитрис. — Второй ключ не будет находиться рядом с первым в таблице. Двоичное дерево поиска тоже не поможет. Из-за того, что ключи *ABCD* и *BCDE* различаются в первом символе, они будут в разных частях дерева.

— Но мы можем использовать для этого суффиксное дерево, — возразил я. — Суффиксное дерево содержит все суффиксы заданного набора строк. Например, суффиксами *ACAC* строки будут {*ACAC*, *CAC*, *AC*, *C*}. Вместе с суффиксами строки *CACT* это даст нам суффиксное дерево, показанное на рис. ЦВ-3.12.

Следуя по указателю из строки *ACAC* на ее самый длинный суффикс *CAC*, мы попадем в правильное место для выполнения проверки строки *CACT* на вхождение в наше множество строк. Значит, нам нужно выполнить только одно сравнение символов.

Суффиксные деревья являются удивительными структурами данных (более подробно они рассматриваются в *разд. 15.3*). Димитрис некоторое время изучал литературу о них, после чего создал элегантный словарь на их основе. Как и прежде, программа выглядела прекрасно, пока мы не испытали ее в действии.

— Теперь скорость работы программы нормальная, но ей не хватает памяти, — пожаловался Димитрис. — Программа создает путь длиной $k$ для каждого суффикса длиной $k$, так что в дереве получается $\Theta(n^2)$ узлов. Когда количество символов превышает 2000, происходит аварийное завершение программы. Мы никогда не сможем дойти до строки из 50 000 символов.

Но я еще не готов был сдаться.

— Проблему с памятью можно решить, используя сжатые суффиксные деревья, — вспомнил я. — Вместо явного представления длинных путей мы можем ссылаться назад на исходную строку. — Сжатые суффиксные деревья всегда занимают линейный объем памяти (см. *разд. 15.3*).

Димитрис снова переделал программу и реализовал структуру данных сжатого суффиксного дерева. Вот *теперь* программа работала отличнейшим образом! Как видно из табл. 3.6, мы смогли без проблем выполнить нашу эмуляцию для строк длиной $n = 65\,536$.

**Таблица 3.6.** Время (в секундах) исполнения эмуляции секвенирования SBH с применением разных структур данных в зависимости от длины строки $n$

| Длина строки | Двоичное дерево | Хеш-таблица | Суффиксное дерево | Сжатое суффиксное дерево |
|---|---|---|---|---|
| 8 | 0,0 | 0,0 | 0,0 | 0,0 |
| 16 | 0,0 | 0,0 | 0,0 | 0,0 |
| 32 | 0,1 | 0,0 | 0,0 | 0,0 |
| 64 | 0,3 | 0,4 | 0,3 | 0,0 |
| 128 | 2,4 | 1,1 | 0,5 | 0,0 |
| 256 | 17,1 | 9,4 | 3,8 | 0,2 |
| 512 | 31,6 | 67,0 | 6,9 | 1,3 |
| 1,024 | 1828,9 | 96,6 | 31,5 | 2,7 |
| 2,048 | 11 441,7 | 941,7 | 553,6 | 39,0 |
| 4,096 | > 2 дней | 5246,7 | нехватка памяти | 45,4 |
| 8,192 | | > 2 дней | | 642,0 |
| 16,384 | | | | 1614,0 |
| 32,768 | | | | 13 657,8 |
| 65,536 | | | | 39 776,9 |

Полученные результаты показали, что интерактивное секвенирование SBH может быть очень эффективным. Это позволило нам заинтересовать биологов нашим методом. Но обеспечение реальных экспериментов в лаборатории стало очередной проверкой нашего знания вычислительных алгоритмов. Описание нашей работы по решению этой задачи приводится в *разд. 12.8*.

Выводы, которые можно сделать, вполне очевидны. Мы выделили одну многократно выполняемую операцию (поиск строки в словаре) и оптимизировали структуру данных для ее поддержки. Когда и улучшенная структура словаря оказалась недостаточно эффективной, мы провели более глубокий анализ выполняемых запросов, чтобы определить, какие дополнительные улучшения следует сделать. Наконец, мы не сдались

после нескольких неудачных попыток, а продолжали поиски подходящей структуры данных, пока не нашли такую, которая обеспечивала бы требуемый уровень производительности. В разработке алгоритмов, как и в жизни, настойчивость обычно приносит результаты.

## Замечания к главе

Оптимизация производительности хеш-таблицы оказывается неожиданно сложной задачей для такой концептуально простой структуры данных. Важность обработки небольших партий элементов при открытой адресации породила более сложные схемы, чем последовательное исследование, для оптимизации работы хеш-таблиц. Подробнее об этом рассказано в книге [Knu98].

На мой взгляд, на хеширование оказал глубокое воздействие доклад выдающегося теоретика Михая Патраску (Mihai Patrascu), который, к сожалению, не дожил до 30 лет. Предмет хеширования и рандомизированных алгоритмов рассматривается более подробно среди прочих в книгах [MR95] и Upfal [MU17].

Программа для оптимизации полосы треугольников, *stripe*, описывается в книге [ESV96]. Использование методов хеширования для выявления плагиата представлено в книге [SWA03].

Обзор алгоритмических вопросов в секвенировании ДНК методом гибридизации приводится в книгах [CK94] и [PL94]. Доклад о нашей работе над интерактивным методом SBH, описанной в *разд. 3.9*, представлен в [MS95a].

## 3.10. Упражнения

### Стеки, очереди и списки

1. *[3]* Распространенной задачей в компиляторах и текстовых редакторах является обеспечение правильного соотношения и вложенности открывающих и закрывающих скобок в строке. Например, в строке (((())())() скобки вложены правильно, а в строках )()( и ()) — нет. Разработайте алгоритм для проверки размещения скобок, который возвращает значение ИСТИНА при правильном вложении скобок и ЛОЖЬ в противном случае. Чтобы решение было полностью засчитано, алгоритм должен определять положение первой неправильной скобки в случае непарных или неправильно вложенных скобок.

2. *[5]* Разработайте алгоритм, который берет в качестве ввода строку $S$, состоящую из открывающих и закрывающих скобок, — например: )()(())(()))))))( — и определяет длину наибольшей последовательности сбалансированных скобок в $S$. Так, в приведенном примере это будет 12. (Подсказка: решение не обязательно будет состоять из последовательности смежных скобок строки $S$.)

3. *[3]* Разработайте алгоритм для изменения направления односвязного списка на противоположное. Иными словами, после обработки программой все указатели должны быть перевернуты в обратном направлении. Алгоритм должен исполняться за линейное время.

4. *[5]* Разработайте стек $S$, поддерживающий операции `S.push(x)`, `S.pop()`, а также операцию `S.findmin()`, которая возвращает наименьший элемент в $S$. Время исполнения всех операций должно быть постоянным.

5. *[5]* Мы видели, как использование динамических массивов позволяет увеличивать размер массива, сохраняя при этом постоянное амортизированное время исполнения. В этой задаче рассматривается вопрос произвольного расширения и уменьшения размеров динамических массивов.

   a) Разработайте стратегию экономии памяти, при которой размер массива, заполненного меньше чем наполовину, уменьшается вдвое. Приведите пример последовательности вставок и удалений, при которой эта стратегия дает неприемлемое амортизированное время исполнения.

   b) Разработайте лучшую стратегию экономии памяти, чем предложенная в пункте (a), в которой достигается постоянное амортизированное время исполнения для каждой операции удаления.

6. *[3]* Предположим, что нужно содержать продукты в холодильнике таким образом, чтобы минимизировать их порчу. Какую структуру данных следует использовать для решения этой задачи и как ее использовать?

7. *[5]* Проработайте детали удаления за постоянное время из односвязного списка, упоминаемого в сноске к *разд. «Остановка для размышлений: Сравнение реализаций словаря (II)»*. В идеале разработайте рабочую реализацию такого удаления. Поддержка других операций должна быть максимально возможно эффективной.

### Элементарные структуры данных

8. *[5]* В игре «Крестики-нолики», которая играется на квадратном поле размером $n \times n$ клеток (обычно $n = 3$), два игрока по очереди ставят в свободные клетки поля символы нуля и крестика (0 и X). Выигрывает игрок, который первым поставит подряд $n$ своих символов в строке, столбце или по диагонали. Создайте структуру данных с пространством $O(n)$, принимающую последовательность ходов и выдающую за постоянное время, выиграл ли игру последний выполненный ход.

9. *[3]* Разработайте функцию, которая принимает последовательность, состоящую из цифр от 2 до 9 и словаря из $n$ слов, и выдает все слова, соответствующие этой последовательности при ее вводе на стандартной телефонной клавиатуре. Например, для последовательности цифр 269 функция должна возвратить, среди прочих, слова *any*, *box*, *boy* и *cow*.

10. *[3]* Строки $X$ и $Y$ являются анаграммами, если буквы строки $X$ можно переупорядочить таким образом, чтобы создавалась строка $Y$. Например, являются анаграммами строки *silent/listen* и *incest/insect*. Разработайте эффективный алгоритм для определения, являются ли строки $X$ и $Y$ анаграммами.

## Деревья и другие словарные структуры

11. *[3]* Разработайте словарную структуру данных с временем исполнения $O(1)$ в наихудшем случае для операций поиска, вставки и удаления. В этой задаче допускается, что элементами множества являются целые числа из конечного множества 1, 2, ..., $n$, а инициализация выполняется за время $O(n)$.

12. *[3]* Максимальная глубина двоичного дерева определяется количеством узлов в пути от корня до наиболее удаленного концевого узла. Разработайте алгоритм с временной

сложностью $O(n)$ для определения максимальной глубины двоичного дерева, содержащего $n$ узлов.

13. *[5]* Два элемента двоичного дерева поиска по ошибке поменяли местами. Разработайте алгоритм с временной сложностью $O(n)$ для обнаружения этих элементов, чтобы их можно было поставить обратно на место.

14. *[5]* Объедините два двоичных дерева поиска в упорядоченный двухсвязный список.

15. *[5]* Опишите алгоритм с временной сложностью $O(n)$, который из двоичного дерева поиска с $n$ узлами создает эквивалентное сбалансированное по высоте двоичное дерево поиска. В сбалансированных двоичных деревьях поиска разница по высоте между левыми и правыми поддеревьями всех узлов всегда не больше 1.

16. *[3]* Определите коэффициент эффективности хранилища (отношение объема памяти, занимаемого данными, к общему объему памяти, отведенному под структуру) для каждой из следующих реализаций двоичного дерева с $n$ узлами:

    a) Все узлы содержат данные, два указателя на дочерние узлы и указатель на родительский узел. Как поле данных, так и каждый указатель занимают четыре байта.

    b) Только листья содержат данные, внутренние узлы содержат два указателя на дочерние узлы. Поле данных занимает четыре байта, а каждый указатель — два байта.

17. *[5]* Разработайте алгоритм с временной сложностью $O(n)$, определяющий, сбалансировано ли заданное двоичное дерево (см. *задачу 15*).

18. *[5]* Опишите, как можно модифицировать любую сбалансированную структуру данных таким образом, чтобы время исполнения операций поиска, вставки, удаления, определения минимума и максимума оставалось равным $O(\log n)$, а операции поиска предшествующего и следующего узлов выполнялись за время $O(1)$. Какие операции нужно модифицировать для этого?

19. *[5]* Допустим, имеется сбалансированный словарь, который поддерживает операции `search`, `insert`, `delete`, `minimum`, `maximum`, `successor` и `predecessor`, время исполнения каждой из которых равно $O(\log n)$. Как можно модифицировать операции вставки и удаления, чтобы их время исполнения оставалось равным $O(\log n)$, но время исполнения операций определения максимума и минимума было $O(1)$. (Подсказка: думайте в терминах абстрактных словарных операций и не теряйте время на указатели и т. п.)

20. *[6]* Разработайте структуру данных, поддерживающую следующие операции:

    - `insert(x,T)` — вставляет элемент $x$ в множество $T$;
    - `delete(k,T)` — удаляет наименьший $k$-й элемент из множества $T$;
    - `member(x,T)` — возвращает значение ИСТИНА тогда и только тогда, когда $x$ является членом $T$.

    Время исполнения всех операций для набора входных данных из $n$ элементов должно быть равным $O(\log n)$.

21. *[8]* Операция конкатенации получает на входе два множества: $S_1$ и $S_2$, где каждый элемент из $S_1$ меньше, чем любой элемент из $S_2$, и соединяет их в одно множество. Разработайте алгоритм для конкатенации двух двоичных деревьев в одно. Время исполнения в наихудшем случае должно быть равно $O(h)$, где $h$ — максимальная высота обоих деревьев.

## Применение древовидных структур

22. *[5]* Разработайте структуру данных, поддерживающую следующие операции:

    - `insert(x)` — вставка элемента $x$ из потока данных в структуру данных;
    - `median()` — возврат медианы всех текущих элементов.

    Время исполнения всех операций для набора входных данных из $n$ элементов должно быть равным $O(\log n)$.

23. *[5]* Предположим, что у нас есть стандартный словарь (сбалансированное двоичное дерево поиска), определенный на множестве из $n$ строк, длина каждой из которых не больше $l$. Нам нужно найти все строки с префиксом $p$. Покажите, как это сделать за время $O(ml \log n)$, где $m$ обозначает количество строк.

24. *[5]* Массив $A$ называется уникальным по $k$, если он не содержит двух одинаковых элементов в пределах $k$ позиций друг от друга. Иными словами, не существует таких значений $i$ и $j$, для которых $A[i] = A[j]$ и $|j - i| \leq k$. Разработайте алгоритм с временной сложностью $O(n \log k)$ в наихудшем случае для проверки, является ли массив $A$ уникальным по $k$.

25. *[5]* В задаче разложения по контейнерам нам нужно разложить $n$ объектов, каждый весом не больше одного килограмма, в наименьшее количество контейнеров, максимальная емкость каждого из которых не больше одного килограмма.

    - Для этого подходит эвристический алгоритм типа «первый лучший». Объекты рассматриваются в порядке, в котором они представлены. Каждый рассматриваемый объект помещается в частично заполненный контейнер, в котором *после* помещения этого объекта останется *наименьший* свободный объем. Если такого контейнера нет, объект помещается в новый (пустой) контейнер. Реализуйте алгоритм «первый лучший» с временем исполнения $O(n \log n)$. Алгоритм принимает в качестве входа множество из $n$ объектов $w_1, w_2, \ldots, w_n$ и возвращает количество требуемых контейнеров.
    - Разработайте и реализуйте алгоритм типа «первый худший», в котором следующий объект помещается в такой частично заполненный контейнер, в котором *после* его помещения останется *наибольший* свободный объем.

26. *[5]* Допустим, что для последовательности из $n$ значений $x_1, x_2, \ldots, x_n$ нам нужно быстро выполнять запросы, в которых, зная $i$ и $j$, нужно найти наименьшее значение в подмножестве $x_i, \ldots, x_j$.

    a) Разработайте структуру данных объемом $O(n^2)$ и временем исполнения запросов $O(1)$.

    b) Разработайте структуру данных объемом $O(n)$ и временем исполнения запросов $O(\log n)$. Чтобы решение было зачтено частично, структура данных может иметь объем $O(n \log n)$ и обеспечивать выполнение запросов за время $O(\log n)$.

27. *[5]* Допустим, что есть множество $S$ из $n$ целых чисел и «черный ящик», который при вводе в него любой последовательности целых чисел и целого числа $k$ немедленно выдает ответ о наличии (или отсутствии) в этой последовательности подмножества, сумма которого равна точно $k$. Покажите, как можно использовать «черный ящик» $O(n)$ раз, чтобы найти подмножество множества $S$, сумма членов которого равна точно $k$.

28. *[5]* Пусть $A[1..n]$ — массив действительных чисел. Разработайте алгоритм для выполнения последовательности следующих операций:

- `add(i,y)` — складывает значение $y$ и $i$-й элемент массива;
- `partial-sum(i)` — возвращает сумму первых $i$ элементов массива, т. е. $\sum_{j=1}^{i} A[j]$.

Нет никаких вставок или удалений, только изменяются значения чисел. Каждая операция должна выполняться за $O(\log n)$ шагов. Можно использовать дополнительный массив размером $n$ в качестве буфера.

29. *[8]* Расширьте структуру данных в предыдущей задаче, чтобы обеспечить поддержку операций вставки и удаления. Каждый элемент теперь имеет *ключ* и *значение*. Доступ к элементу осуществляется по его ключу, но операция сложения выполняется над значениями элементов. Операция `partial-sum` отличается от аналогичной операции из предыдущей задачи. Используемые операции таковы:
    - `add(k,y)` — складывает значение $y$ со значением элемента с ключом $k$;
    - `insert(k,y)` — вставляет новый элемент с ключом $k$ и значением $y$;
    - `delete(k)` — удаляет элемент с ключом $k$;
    - `partial-sum(k)` — возвращает сумму всех текущих элементов множества, у которых значение ключа меньше чем $k$, т. е. $\sum_{i<k} x_i$.

    Время исполнения в наихудшем случае для любой последовательности $O(n)$ операций должно оставаться таким же — $O(n \log n)$.

30. *[8]* Вы предоставляете услуги консультанта гостинице, которая содержит $n$ комнат с одной кроватью. Прибывающий постоялец запрашивает комнату с номером в диапазоне $[l, h]$. Разработайте структуру данных, поддерживающую выполнение следующих операций за указанное время:

    a) `Initialize(n)` — инициализирует структуру данных для пустых комнат с номерами 1, 2, ..., $n$ за полиномиальное время;

    b) `Count(l, h)` — возвращает количество свободных комнат в диапазоне $[l, h]$ за время $O(\log n)$;

    c) `Checkin(l, h)` — возвращает первую свободную комнату в диапазоне $[l, h]$ и помечает ее как занятую, или же возвращает NIL, если все комнаты в этом диапазоне заняты. Время исполнения — $O(\log n)$;

    d) `Checkout(x)` — обозначает комнату $x$ как незанятую за время $O(\log n)$.

31. *[8]* Разработайте структуру данных, поддерживающую операции поиска, вставки и удаления целого числа $X$ за время $O(1)$ (т. е. за постоянное время, независимо от общего количества хранящихся в структуре целых чисел). Допустим, что $1 < X < n$ и что существует $m + n$ ячеек для хранения целых чисел, где $m$ — наибольшее количество целых чисел, которые могут одновременно храниться в структуре. (Подсказка: используйте два массива: $A[1..n]$ и $B[1..m]$.) Массивы нельзя инициализировать, поскольку для этого понадобится $O(m)$ или $O(n)$ операций. Отсюда следует, что массивы изначально содержат непредсказуемые значения, поэтому вам нужно проявлять особую аккуратность.

## Задачи по реализации

32. *[5]* Реализуйте разные варианты словарных структур данных, такие как связные списки, двоичные деревья, сбалансированные двоичные деревья поиска и хеш-таблицы. Экспе-

риментальным путем сравните производительность этих структур данных в простом приложении, считывающем текстовый файл большого объема и отмечающем только один раз каждое встречающееся в нем слово. Это приложение можно эффективно реализовать, создав словарь и вставляя в него каждое новое обнаруженное в тексте слово. Напишите краткий доклад с вашими выводами.

33. *[5]* Шифр Цезаря (см. *разд. 21.6*) относится к классу простейших шифров. К сожалению, зашифрованные таким способом сообщения можно расшифровать, используя статистические данные о частотном распределении букв латинского алфавита. Разработайте программу для расшифровки достаточно длинных текстов, зашифрованных шифром Цезаря.

## Задачи, предлагаемые на собеседовании

34. *[3]* Каким методом вы бы воспользовались, чтобы найти слово в словаре?

35. *[3]* Представьте, что у вас полный шкаф рубашек. Как можно организовать рубашки, чтобы упростить их извлечение из шкафа?

36. *[4]* Напишите функцию поиска среднего узла односвязного списка.

37. *[4]* Напишите функцию для определения, идентичны ли два двоичных дерева. Идентичные двоичные деревья имеют одинаковую структуру и одинаковые значения в соответствующих узлах.

38. *[4]* Напишите программу для преобразования двоичного дерева поиска в связный список.

39. *[4]* Реализуйте алгоритм для изменения направления связного списка на обратное. Разработайте такой же алгоритм, но без использования рекурсии.

40. *[5]* Какой тип структуры данных будет наилучшим для хранения интернет-адресов (URL), посещенных поисковым механизмом? Создайте алгоритм для проверки, был ли ранее посещен тот или иной URL, оптимизируйте алгоритм по времени и памяти.

41. *[4]* У вас имеется строка поиска и журнал. Вам нужно сгенерировать все символы в строке поиска, выбрав их из журнала. Создайте эффективный алгоритм для определения, содержит ли журнал все символы в строке поиска.

42. *[4]* Создайте алгоритм для изменения порядка слов в предложении на обратный, т. е. фраза «Меня зовут Крис» должна превратиться в «Крис зовут меня». Оптимизируйте алгоритм по времени и памяти.

43. *[5]* Разработайте как можно более быстрый алгоритм, использующий минимальный объем памяти, для определения, содержит ли связный список петлю. В положительном случае определите местонахождение петли.

44. *[5]* Имеется неотсортированный массив $X$, содержащий $n$ целых чисел. Определите множество $M$, содержащее $n$ элементов, где $M_i$ является произведением всех целых чисел из $X$, за исключением $X_i$. Операцию деления применять нельзя, но можно использовать дополнительную память. (Подсказка: существуют решения с временем исполнения быстрее, чем $O(n^2)$.)

45. *[6]* Разработайте алгоритм, позволяющий найти на веб-странице наиболее часто встречающуюся упорядоченную пару слов (например, «New York»). Какой тип структуры данных вы бы использовали? Оптимизируйте алгоритм по времени и памяти.

## LeetCode

1. **https://leetcode.com/problems/validate-binary-search-tree/**
2. **https://leetcode.com/problems/count-of-smaller-numbers-after-self/**
3. **https://leetcode.com/problems/construct-binary-tree-from-preorder-andinorder-traversal/**

## HackerRank

1. **https://www.hackerrank.com/challenges/is-binary-search-tree/**
2. **https://www.hackerrank.com/challenges/queue-using-two-stacks/**
3. **https://www.hackerrank.com/challenges/detect-whether-a-linked-list-contains-a-cycle/problem**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org:**

1. «Jolly Jumpers», глава 2, задача 10038.
2. «Crypt Kicker», глава 2, задача 843.
3. «Where's Waldorf?», глава 3, задача 10010.
4. «Crypt Kicker II», глава 3, задача 850.

# ГЛАВА 4

# Сортировка и поиск

Студенты, получающие специальность, связанную с вычислительными системами, изучают основные алгоритмы сортировки по крайней мере три раза: сначала во введении в программирование, затем в курсе по структурам данных и, наконец, в курсе разработки и анализа алгоритмов. Почему сортировке уделяется так много внимания? На то есть ряд причин.

- Сортировка является базовым строительным блоком, на котором основаны многие алгоритмы. Понимание сортировки расширяет наши возможности при решении других задач.
- Большинство интересных идей, которые используются в разработке алгоритмов (в частности, метод «разделяй и властвуй», структуры данных и рандомизированные алгоритмы), возникли в контексте сортировки.
- Сортировка — самая изученная задача в теории вычислительных систем. Известны буквально десятки алгоритмов сортировки, большинство из которых имеют определенное преимущество над другими алгоритмами в определенных ситуациях.

В этой главе мы обсудим сортировку, уделяя особое внимание тому, как ее можно применить для решения других задач. Мы рассмотрим подробное описание нескольких фундаментальных алгоритмов: *пирамидальной сортировки*[1] (heapsort), *сортировки слиянием* (mergesort), *быстрой сортировки* (quicksort) и *сортировки распределением* (distribution sort), — представляющих собой примеры важных парадигм разработки алгоритмов. Задача сортировки также представлена в *разд. 17.1*.

## 4.1. Применение сортировки

В этой главе мы рассмотрим несколько алгоритмов сортировки и оценим их временну́ю сложность. Важнейшая идея, которую я хочу донести до читателя, заключается в том, что существуют алгоритмы сортировки со временем исполнения $O(n \log n)$. Это намного лучше, чем производительность $O(n^2)$, которую демонстрируют простые алгоритмы сортировки на больших значениях $n$.

В табл. 4.1 приведен пример количества шагов, выполняемых двумя разными алгоритмами сортировки для разумных значений $n$.

Алгоритмы квадратичной сложности могут быть приемлемыми еще при $n = 10\,000$, но когда $n > 100\,000$, медленный алгоритм сортировки за квадратичное время становится неприемлемым.

---

[1] Другое название: «сортировка кучей». — *Прим. пер.*

**Таблица 4.1.** Пример количества шагов, выполняемых двумя разными алгоритмами сортировки

| $n$ | $n^2/4$ | $n \lg n$ |
|---|---|---|
| 10 | *25* | 33 |
| 100 | 2500 | *664* |
| 1000 | 250 000 | *9965* |
| 10 000 | 25 000 000 | *132 877* |
| 100 000 | 2 500 000 000 | *1 660 960* |

Большинство важных задач можно свести к сортировке — в результате задачу, на первый взгляд требующую квадратичного алгоритма, можно решить с помощью интеллектуальных алгоритмов с временно́й сложностью $O(n \log n)$. Одним из важных методов разработки алгоритмов является использование сортировки в качестве базового конструктивного блока, т. к. после сортировки набора входных данных многие другие задачи становятся легко решаемыми.

Рассмотрим несколько задач.

- *Поиск.*

  При условии, что входные данные отсортированы, двоичный поиск элемента в словаре занимает время $O(\log n)$. Предварительная обработка данных для поиска, пожалуй, является наиболее важным применением сортировки.

- *Поиск ближайшей пары.*

  Как найти в множестве из $n$ чисел два числа с наименьшей разностью между ними? После сортировки входного набора такие числа должны в упорядоченной последовательности находиться рядом. Поиск этих чисел занимает линейное время, а общее время, включая сортировку, составляет $O(n \log n)$.

- *Определение уникальности элементов.*

  Как определить, имеются ли дубликаты в множестве из $n$ элементов? Это частный случай общей задачи поиска ближайшей пары элементов, но сейчас нам нужно найти два элемента, разность между которыми равна нулю. Эта задача решается так же, как и предыдущая: сначала выполняется сортировка входного множества, после чего отсортированная последовательность перебирается за линейное время, которое требуется для проверки всех смежных пар.

- *Определение моды.*

  Задача состоит в определении наиболее часто встречающегося элемента в множестве из $n$ элементов. Если элементы множества отсортированы, то их можно просто перебрать слева направо, подсчитывая, сколько раз встречается каждый элемент, поскольку при сортировке все одинаковые элементы будут размещены рядом друг с другом.

  Для подсчета количества вхождений произвольного элемента $k$ в некоторое множество выполняется двоичный поиск этого элемента в отсортированном массиве клю-

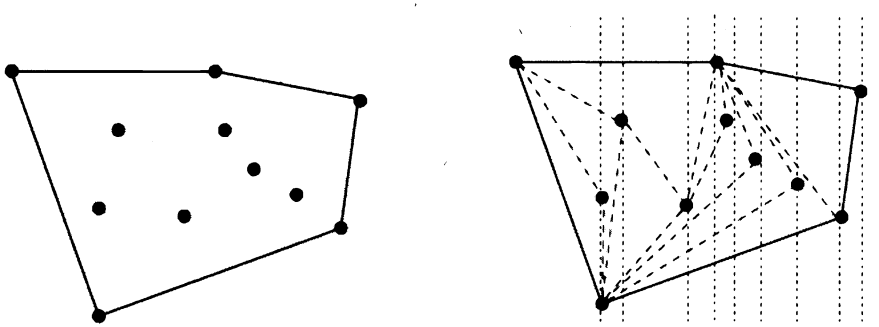
чей. После нахождения требуемого элемента в массиве выполняется сканирование влево от этого элемента, пока не будет найден первый элемент, отличный от $k$. Потом эта же процедура повторяется вправо от первоначальной точки. Общее время определения вхождений этим методом равно $O(\log n + c)$, где $c$ — количество вхождений элемента $k$. Еще лучшее время — $O(\log n)$ — можно получить, определив посредством двоичного поиска местонахождение как элемента $k - \varepsilon$, так и элемента $k + \varepsilon$ (где $\varepsilon$ — достаточно малое), и затем вычислив разницу между этими двумя позициями.

- *Выбор элемента.*

  Как найти $k$-й по величине элемент массива? Если элементы массива отсортированы, то $k$-й по величине элемент можно найти за линейное время, поскольку он должен находиться в $k$-й ячейке массива. В частности, средний элемент (см. *разд. 17.3*) находится в отсортированном массиве в $(n/2)$-й позиции.

- *Выпуклая оболочка.*

  Как определить многоугольник с наименьшим периметром, содержащий множество точек $n$ в двух измерениях? Выпуклую оболочку можно сравнить с резиновой нитью, натянутой вокруг точек в плоскости. Резиновая нить сжимается только вокруг самых выступающих точек, образуя многоугольник (рис. 4.1, *а*).



*а* *б*

**Рис. 4.1.** Создание выпуклой оболочки: резиновой нитью *(а)*, вставкой точек слева направо *(б)*

Выпуклая оболочка дает хорошее представление о размещении точек и является одним из важных конструктивных блоков для создания более сложных геометрических алгоритмов, рассматриваемых в *разд. 20.2*.

Но каким образом можно использовать сортировку для создания выпуклой оболочки? После того как точки отсортированы по $x$-координате, их можно вставлять в оболочку слева направо. Так как самая правая точка всегда расположена на периметре, то мы знаем, что она должна войти в оболочку. После добавления этой новой самой правой точки остальные могут оказаться удаленными, но мы легко идентифицируем эти точки, поскольку они находятся внутри многоугольника, полученного в результате добавления новой точки (рис. 4.1, *б*). Эти точки будут соседями предыдущей вставленной точки, так что их можно будет с легкостью найти и удалить. После сортировки общее время исполнения — линейное.

Хотя некоторые из этих задач (например, выбор элемента) можно решить за линейное время с помощью более сложных алгоритмов, сортировка предоставляет самые простые решения. Время исполнения сортировки оказывается узким местом лишь в немногих приложениях, к тому же в этом случае ее почти всегда можно заменить на более интеллектуальные алгоритмы. Поэтому никогда не бойтесь потратить время на сортировку при условии, что используется эффективная процедура сортировки.

#### Подведение итогов

Сортировка является центральной частью многих алгоритмов. Сортировка данных должна быть одним из первых шагов, предпринимаемых любым разработчиком алгоритмов с целью повышения эффективности разрабатываемого решения.

### Остановка для размышлений: Поиск пересечения множеств

**ЗАДАЧА**. Предоставить эффективный алгоритм для определения, являются ли два множества (мощностью $m$ и $n$ соответственно) непересекающимися. Проанализировать сложность алгоритма в наихудшем случае относительно $m$ и $n$, рассматривая случай, когда $m << n$.

**РЕШЕНИЕ**. В голову приходят по крайней мере три решения, каждое из которых является разновидностью сортировки и поиска.

- *Сортируется только большое множество.* Большое множество можно отсортировать за время $O(n \log n)$. Теперь для каждого из $m$ элементов меньшего множества выполняется поиск на его наличие в большом множестве. Общее время исполнения будет равно $O((n + m)\log n)$.
- *Сортируется только меньшее множество.* Меньшее множество сортируется за время $O(m \log m)$. Теперь для каждого из $n$ элементов большего множества выполняется поиск на его наличие в меньшем множестве. Общее время исполнения будет равно $O((n + m)\log n)$.
- *Сортируются оба множества.* Когда оба множества отсортированы, то для определения общего элемента больше не требуется выполнять двоичный поиск. Вместо этого мы сравниваем наименьшие элементы в обеих отсортированных последовательностях и, если они различаются, удаляем меньший элемент. Операция повторяется рекурсивно на все уменьшающихся последовательностях, занимая линейное время, а общее время исполнения (включая сортировку) равно $O(n \log n + m \log m + n + m)$.

Какой же из этих методов самый быстрый? Ясно, что сортировка меньшего множества выполняется быстрее, чем сортировка большего множества, т. к. $\log m < \log n$ при $m < n$. Аналогично $(n + m)\log m$ должно быть асимптотически меньше, чем $n \log n$, т. к. $n + m < 2n$ при $m < n$. Следовательно, метод сортировки меньшего множества является самым лучшим из этих трех. Обратите внимание, что при постоянном значении $m$ время исполнения будет линейным.

Кроме этого, *ожидаемое* линейное время можно получить посредством хеширования: создается хеш-таблица, содержащая элементы обоих множеств, после чего выполняется явная проверка, что конфликты в корзине действительно являются результатом

хеширования идентичных элементов. На практике это решение может быть самым лучшим. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Использование хеша для решения задач

**ЗАДАЧА**. Алгоритм быстрой сортировки, несомненно, прекрасная вещь. Но какую из следующих задач можно выполнить за такое же (ожидаемое) время, или даже быстрее, используя хеширование вместо сортировки?

- Поиск?
- Поиск ближайшей пары?
- Определение уникальности элементов?
- Определение моды?
- Определение медианы?
- Выпуклая оболочка?

**РЕШЕНИЕ**. Некоторые из этих задач можно эффективно решить хешированием, но для других этот подход неприемлем. Рассмотрим каждую из задач по отдельности:

- *Поиск* — таблицы хеширования прекрасно подходят для решения этой задачи, позволяя выполнить поиск элементов за постоянное ожидаемое время, в противоположность времени $O(\log n)$ для двоичного поиска.
- *Поиск ближайшей пары* — рассмотренные к этому моменту таблицы хеширования абсолютно непригодны для решения такой задачи. Обычные функции хеширования разбрасывают ключи по всей таблице, поэтому маловероятно, что пара подобных числовых значений окажется в одной и той же корзине. Распределение по корзинам диапазонов значений обеспечит помещение пары ближайших значений в одну корзину или в наихудшем случае в соседние корзины. Но мы не можем принудить помещение в корзину только небольшого количества элементов, как можно видеть в рассмотрении корзинной сортировки в *разд. 4.7*.
- *Определение уникальности элементов* — эта задача выполняется быстрее при использовании хеширования, чем простой сортировкой. Для этого создаем таблицу хеширования посредством метода цепочек, а затем сравниваем каждую из пар элементов в корзине. Если ни одна из корзин не содержит пары дубликатов, тогда все элементы должны быть однозначными. Создание таблицы и сканирование элементов в ней можно выполнить за ожидаемое линейное время.
- *Определение моды* — использование хеширования для решения этой задачи позволяет получить алгоритм с ожидаемым линейным временем исполнения. Каждая корзина должна содержать небольшое количество *различных* элементов, но может иметь много дубликатов. Начиная с первого элемента в корзине, подсчитываем и удаляем все его копии, повторяя эту операцию ожидаемое постоянное количество раз, пока корзина не опустеет.
- *Определение медианы* — к сожалению, здесь хеширование нам не поможет. Медиана может находиться в любой корзине таблицы, и у нас нет никакого способа судить о количестве элементов, которое может быть до или после него в упорядоченном списке.

- *Выпуклая оболочка* — конечно же, по точкам можно создать таблицу хеширования, как и любой другой тип данных. Но не совсем ясно, какую пользу она принесет нам для решения этой задачи, и, несомненно, не поможет нам упорядочить точки по координате $x$. ■

# 4.2. Практические аспекты сортировки

Мы уже видели, что сортировка часто применяется в различных алгоритмах. Теперь обсудим несколько эффективных алгоритмов сортировки. Это делает актуальным вопрос: в каком порядке нужно сортировать элементы?

Ответ на этот вопрос зависит от конкретной задачи. Более того, во внимание должен приниматься ряд соображений.

- *Сортировать в возрастающем или убывающем порядке?*

  Набор ключей $S$ отсортирован в возрастающем порядке, если $S_i \leq S_{i-1}$ для всех $1 \leq i < n$, и в убывающем порядке, если $S_i \geq S_{i+1}$ для всех $1 \leq i < n$. Для разных приложений требуется разный порядок сортировки.

- *Сортировать только ключ или все поля записи?*

  При сортировке набора данных необходимо сохранять целостность сложных записей. Например, элементы списка рассылки, содержащего имена, адреса и телефонные номера, можно отсортировать по полю имен, но при этом необходимо сохранять связь между полем имен и другими полями. Таким образом, для сложной записи нам нужно указать, какое поле является ключевым, а также понимать полную структуру записей.

- *Что делать в случае совпадения ключей?*

  Элементы с одинаковыми значениями ключей будут сгруппированы вместе при любом порядке сортировки, но иногда имеет значение порядок размещения этих элементов относительно друг друга. Допустим, что энциклопедия содержит записи, как для Майкла Джордана баскетболиста, так и для Майкла Джордана актера[2]. Какая из этих записей должна быть первой? Для решения вопроса одинаковых первичных ключей придется прибегнуть к использованию вторичного ключа, которым может оказаться, например, размер статьи.

  Иногда при сортировке необходимо сохранить первоначальный порядок элементов с одинаковыми ключами. Алгоритмы сортировки, которые автоматически выполняют это требование, называются *устойчивыми* (stable). Но устойчивыми являются очень немногие быстрые алгоритмы. Стабильность в любых алгоритмах сортировки можно обеспечить, указав первоначальное положение записи в качестве вторичного ключа.

  Конечно же, в случае совпадения ключей можно ничего не делать, а просто позволить алгоритму разместить их там, где он сочтет нужным. Но имейте в виду, что производительность некоторых эффективных алгоритмов сортировки (например,

[2] Не говоря уже о Майкле Джордане, специалисте в области статистики.

быстрой сортировки) может ухудшиться до квадратичной, если в них явно не предусмотреть возможность обработки большого количества одинаковых значений ключей.

- *Как поступать с нечисловыми данными?*

  Сортировка текстовых строк определяет их упорядочение по алфавиту. В библиотеках применяются очень подробные и сложные правила касательно последовательности упорядочения букв и знаков пунктуации. Это вызвано необходимостью принимать такие решения, как, например, являются ли ключи `Skiena` и `skiena` одинаковыми, или как упорядочить записи `Brown-Williams`, `Brown America` и `Brown, John`?

Правильным решением таких тонкостей в алгоритме сортировки будет использование *функции попарного сравнения* элементов, специфической для конкретного приложения. Эта функция принимает в качестве входа указатели на элементы $a$ и $b$ и возвращает «<», если $a < b$, «>», если $a > b$, и «=», если $a = b$. Сводя попарное упорядочивание к подобной функции, мы можем реализовывать алгоритмы сортировки, не обращая внимания на такие детали. Мы просто передаем функцию сравнения процедуре сортировки в качестве аргумента. Любой серьезный язык программирования содержит встроенную в виде библиотечной функции процедуру сортировки. Обычно использование этой функции предпочтительнее написания своей собственной процедуры сортировки. Например, стандартная библиотека языка C содержит функцию сортировки `qsort`[3]:

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
           int (*compare) (const void *, const void *));
```

При использовании функции `qsort` важно понимать назначение ее аргументов. Функция сортирует первые `nel` элементов массива, длина которых составляет `width` байтов (на сам массив указывает переменная `base`). То есть мы можем сортировать массивы однобайтовых символов, четырехбайтовых целых чисел или 100-байтовых записей, изменяя лишь значение переменной `width`.

Требуемый порядок отсортированной последовательности определяется функцией `compare`. Эта функция принимает в качестве аргументов указатели на два элемента размером `width` и возвращает отрицательное число, если в отсортированной последовательности первый элемент должен быть перед вторым, положительное число, если наоборот, и ноль, если элементы одинаковые. Код функции для сравнения целых чисел в возрастающей последовательности приведен в листинге 4.1.

**Листинг 4.1. Реализация функции сравнения**

```
int intcompare(int *i, int *j)
{
   if (*i > *j) return (1);
```

[3] Фрагменты кода, не оформленные в тексте как листинги, в файле листингов с цветными элементами (см. *приложение*) помечены как «Код» — например, этот фрагмент помечен так: «Код 4.1». Последовательность размещения таких фрагментов в файле листингов соответствует размещению их в текстах глав. — *Прим. ред.*

```
    if (*i < *j) return (-1);
    return (0);
}
```

Эту функцию сравнения можно использовать при сортировке массива a, в котором заняты первые n ячеек. Функция сортировки вызывается таким образом:

```
qsort(a, n, sizeof(int), intcompare);
```

Название функции сортировки qsort дает основания полагать, что в ней реализован алгоритм быстрой сортировки quicksort, но это обстоятельство обычно не имеет никакого значения для пользователя.

# 4.3. Пирамидальная сортировка

Тема сортировки представляет собой естественную лабораторию для изучения принципов разработки алгоритмов, т. к. использование различных методов сортировки влечет за собой создание интересных ее алгоритмов. В следующих нескольких разделах представляются методы разработки алгоритмов, порожденные определенными алгоритмами сортировки.

Внимательный читатель может спросить, почему мы рассматриваем стандартные методы сортировки, когда в конце предыдущего раздела была дана рекомендация обычно не реализовывать их, а вместо этого использовать библиотечные функции сортировки конкретного языка программирования. Ответ на этот вопрос состоит в том, что фундаментальные методы, применяемые для разработки этих алгоритмов, также являются очень важными для разработки алгоритмов решения других задач, с которыми вам, скорее всего, придется столкнуться.

Мы начнем с разработки структур данных, поскольку одним из методов существенного улучшения производительности алгоритмов сортировки является использование соответствующих задаче структур данных. Сортировка методом выбора представляет собой простой алгоритм, который в цикле извлекает наименьший оставшийся элемент из неотсортированной части набора входных данных. На псевдокоде этот алгоритм можно выразить таким образом:

```
SelectionSort(A)
     For i = 1 to n do
          Sort[i] = Find-Minimum from A
          Delete-Minimum from A
     Return(Sort)
```

Реализация алгоритма на языке C приводится в листинге 2.1. Там массив входных данных разделяется на отсортированную и неотсортированную части. Поиск наименьшего элемента выполняется сканированием элементов в неотсортированной части массива, что занимает линейное время. Найденный наименьший элемент меняется местами с элементом $i$ массива, после чего цикл повторяется. Всего выполняется $n$ итераций, в каждой из которых в среднем $n/2$ шагов, а общее время исполнения равно $O(n^2)$.

Может быть, существует лучшая структура данных? Когда местонахождение элемента в неотсортированном массиве определено, его удаление занимает время $O(1)$, но чтобы

найти наименьший элемент, нужно время $O(n)$. Эти операции могут использовать очереди с приоритетами. Что будет, если заменить текущую структуру данных структурой с реализацией очереди с приоритетами — такой как пирамида или сбалансированное двоичное дерево? Тогда вместо $O(n)$ операции внутри цикла исполняются за время $O(\log n)$. Использование такой реализации очереди с приоритетами ускоряет сортировку методом выбора с $O(n^2)$ до $O(n \log n)$.

Распространенное название этого алгоритма — *пирамидальная сортировка* — скрывает тот факт, что пирамидальная сортировка в действительности — это не что иное, как реализация сортировки методом выбора с применением удачной структуры данных.

## 4.3.1. Пирамиды

*Пирамида* представляет собой простую и элегантную структуру данных, поддерживающую такие операции очередей с приоритетами, как вставка и поиск наименьшего элемента. Принцип работы пирамиды основан на поддержании порядка «частичной отсортированности» в заданном наборе элементов. Такой порядок слабее, чем состояние полностью отсортированного набора (что обеспечивает эффективность сопровождения), но сильнее, чем произвольный порядок (что обеспечивает быстрый поиск наименьшего элемента).

Властные отношения в любой организации с иерархической структурой отображаются деревом, каждый узел которого представляет сотрудника, а ребро $(x, y)$ означает, что $x$ непосредственно управляет $y$ (доминирует над ним). Сотрудник, отображаемый корневым узлом, находится наверху организационной пирамиды.

Подобным образом *пирамида* определяется как двоичное дерево, в котором значение ключа каждого узла *доминирует* над значением ключа каждого из его потомков. В *неубывающей* бинарной пирамиде (min-heap) доминирующим над своими потомками является узел с меньшим значением ключа, чем значения ключей его потомков; а в *невозрастающей* бинарной пирамиде (max-heap) доминирующим является узел со значением ключа большим, чем значения ключей его потомков. На рис. 4.2, *слева* показана неубывающая пирамида дат знаменательных событий в истории Соединенных Штатов.

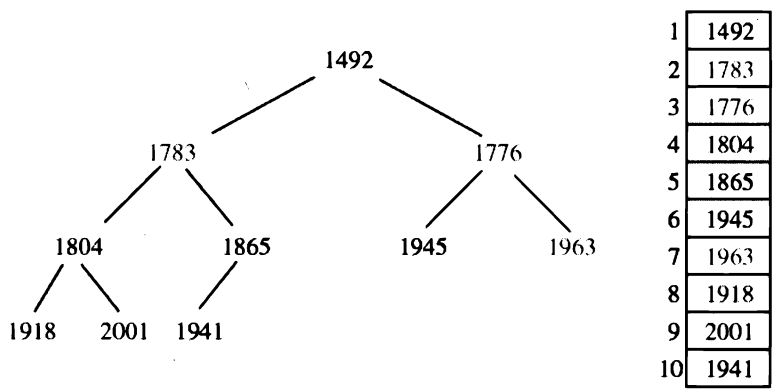



**Рис. 4.2.** Пирамидальное дерево дат важных событий в американской истории (*слева*) и соответствующий массив (*справа*)

В наиболее естественной реализации этого двоичного дерева каждый ключ сохранялся бы в узле с указателями на двух его потомков. Но, как и в случае с двоичными деревьями поиска, объем памяти, занимаемый указателями, может быстро превысить объем памяти, занимаемый ключами, которые интересуют нас в первую очередь.

Однако пирамида является настолько удачной структурой данных, что позволяет нам реализовать двоичные деревья без помощи указателей. Данные в ней сохраняются в виде массива ключей, а позиции ключей используются в роли *неявных* указателей:

```
typedef struct {
     item_type q[PQ_SIZE+1];    /* Тело очереди */
     int n;                     /* Количество элементов очереди */
} priority_queue;
```

Корневой узел дерева сохраняется в первой ячейке массива, а его левый и правый потомки — во второй и третьей соответственно. В общем, мы сохраняем $2^{l-1}$ ключей уровня $l$ полного двоичного дерева в направлении слева направо в позициях $2^{l-1}$ до $2^l - 1$, как показано на рис. 4.2, *справа*. Для простоты мы полагаем, что нумерация ячеек массива начинается с единицы.

Это представление весьма удачно благодаря легкости, с которой определяется местонахождение родителя и потомка ключа, расположенного в позиции $k$. Левый потомок ключа $k$ находится в позиции $2k$, правый — в позиции $2k + 1$, а родительский ключ находится в позиции $\lfloor k/2 \rfloor$. Таким образом, по дереву можно перемещаться без использования указателей (листинг 4.2).

**Листинг 4.2. Код для работы с пирамидой**

```
int pq_parent(int n) {
    if (n == 1) {
        return(-1);
    }
    return((int) n/2); /* явно вычисляем floor(n/2) */
}

int pq_young_child(int n) {
    return(2 * n);
}
```

Этот подход означает, что мы можем сохранить любое двоичное дерево в массиве, не используя указатели. Но нет ли здесь какого-то подвоха? Да, есть. Допустим, что наше дерево высотой $h$ разреженное, т. е. количество его узлов $n << 2^h - 1$. Тем не менее нам нужно выделить место в структуре для всех отсутствующих узлов, поскольку необходимо представить полное двоичное дерево, чтобы сохранить позиционные отношения между узлами.

При этом, чтобы не расходовать понапрасну память, мы не можем допустить наличия пробелов в нашем дереве — иными словами, каждый уровень должен быть заполнен до предела. Тогда только последний уровень может быть заполнен частично. Упаковывая элементы последнего уровня как можно плотнее влево, мы можем представить де-

рево из $n$ ключей, используя первые $n$ элементов массива. Если не придерживаться этих структурных ограничений, то для представления $n$ элементов нам может потребоваться массив размером $2n - 1$ (рассмотрим идущий вправо концевой узел с позициями 1, 3, 7, 15, 31...). Так как в пирамидальном дереве всегда заполняются все уровни, кроме последнего, то высота $h$ пирамиды из $n$ элементов логарифмическая, поскольку

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1 \geq n,$$

а это означает, что $h = \lceil \lg(n+1) \rceil$.

Хотя такое неявное представление двоичных деревьев позволяет сэкономить память, оно менее гибкое, чем представление с использованием указателей. Сохранение деревьев с произвольной топологией влечет за собой большой расход памяти. Кроме этого, размещение поддеревьев можно изменять, только явно перемещая все элементы поддерева, а не обновляя один-единственный указатель, как это делается в случае деревьев с указателями. Такое отсутствие гибкости объясняет, почему эту идею нельзя использовать для представления двоичных деревьев поиска. Впрочем, для пирамид она вполне подходит.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Поиск в пирамиде

**ЗАДАЧА**. Возможен ли эффективный поиск определенного ключа $k$ в пирамиде?

**РЕШЕНИЕ**. Невозможен. В пирамиде нельзя использовать двоичный поиск, т. к. пирамида не является двоичным деревом. Нам почти ничего не известно об относительном размещении $n/2$ листьев пирамиды — во всяком случае ничего, что помогло бы избежать линейного поиска в этих листьях. ■

## 4.3.2. Создание пирамиды

Пирамиду можно создать пошагово, вставляя каждый новый элемент в самую левую свободную ячейку массива, а именно в $(n+1)$-ю ячейку предыдущей пирамиды из $n$ элементов. Таким образом обеспечивается сбалансированная форма пирамидального дерева, но при этом не соблюдается порядок доминирования ключей. Новый ключ может быть меньшим, чем его предшественник в неубывающей бинарной пирамиде, или большим, чем его предшественник в невозрастающей бинарной пирамиде.

Эта задача решается за счет обмена местами такого элемента с его родителем: прежний родитель занимает должное место в иерархии доминирования, а порядок доминирования другого дочернего узла прежнего родителя продолжает оставаться правильным, поскольку теперь над ним находится элемент с бóльшим уровнем доминирования, чем у предыдущего. Новый элемент оказывается в несколько лучшем положении, но может продолжать доминировать над своим новым родителем. Поэтому мы повторяем процедуру необходимое количество раз на более высоком уровне, перемещая новый ключ на должное место в иерархии *пузырьковым методом*. Так как на каждом шаге корень поддерева заменяется элементом с большим уровнем доминирования, то порядок доминирования сохраняется во всех других частях пирамиды.

Код для вставки нового элемента в пирамиду приведен в листинге 4.3.

**Листинг 4.3. Вставка элемента в пирамиду**

```
void pq_insert(priority_queue *q, item_type x) {
    if (q->n >= PQ_SIZE) {
        printf("Warning: priority queue overflow! \n");
    } else {
        q->n = (q->n) + 1;
        q->q[q->n] = x;
        bubble_up(q, q->n);
    }
}

void bubble_up(priority_queue *q, int p) {
    if (pq_parent(p) == -1) {
        return; /* Корень */
    }

    if (q->q[pq_parent(p)] > q->q[p]) {
        pq_swap(q, p, pq_parent(p));
        bubble_up(q, pq_parent(p));
    }
}
```

На каждом уровне процесс обмена элементов местами занимает постоянное время. Так как высота пирамиды из $n$ элементов равна $\lfloor \lg n \rfloor$, то каждая вставка занимает самое большее время $O(\log n)$. Таким образом, первоначальную пирамиду из $n$ элементов можно создать посредством $n$ таких вставок за время $O(n \log n)$. Соответствующий код приведен в листинге 4.4.

**Листинг 4.4. Создание пирамиды повторяющимися вставками**

```
void pq_init(priority_queue *q) {
    q->n = 0;
}

void make_heap(priority_queue *q, item_type s[], int n) {
    int i; /* счетчик */

    pq_init(q);
    for (i = 0; i < n; i++) {
        pq_insert(q, s[i]);
    }
}
```

## 4.3.3. Наименьший элемент пирамиды

Осталось рассмотреть такие операции, как поиск и удаление корневого элемента пирамиды. Поиск не составляет никакого труда, поскольку верхний элемент пирамиды находится в первой ячейке массива.

После удаления элемента в массиве остается пустая ячейка, которую можно заполнить, перемещая в нее элемент из самого *правого* листа (который размещен в *n*-й ячейке массива).

Таким образом восстанавливается форма дерева, но (как и в случае со вставкой) значение корневого узла может больше не удовлетворять иерархическим требованиям пирамиды. Более того, над этим новым корнем могут доминировать оба его потомка. Корень такой неубывающей бинарной пирамиды должен быть наименьшим из этих трех элементов, т. е. этого корня и двух его потомков. Если текущий корень доминирует над своими потомками, значит, порядок пирамиды восстановлен. В противном случае доминирующий потомок меняется местами с корнем, и проблема передается на один уровень вниз.

Проблемный элемент продвигается вниз *пузырьковым методом* до тех пор, пока он не начнет доминировать над всеми своими потомками, возможно, став листом. Такая операция «просачивания» элемента вниз также называется *восстановлением пирамиды* (heapify), поскольку она сливает вместе две пирамиды (поддеревья под первоначальным корнем) с новым ключом. Код для удаления наименьшего элемента пирамиды приведен в листинге 4.5.

**Листинг 4.5. Удаление наименьшего элемента пирамиды**

```
item_type extract_min(priority_queue *q) {
    int min = -1; /* Минимальное значение  */

    if (q->n <= 0) {
        printf("Warning: empty priority queue.\n");
    } else {
        min = q->q[1];

        q->q[1] = q->q[q->n];
        q->n = q->n - 1;
        bubble_down(q, 1);
    }
    return(min);
}

void bubble_down(priority_queue *q, int p) {
    int c; /* Индекс потомка */
    int i; /* Счетчик */
    int min_index; /* Индекс наименьшего потомка */

    c = pq_young_child(p);
    min_index = p;

    for (i = 0; i <= 1; i++) {
        if ((c + i) <= q->n) {
            if (q->q[min_index] > q->q[c + i]) {
                min_index = c + i;
            }
        }
    }
```

```
    if (min_index != p) {
        pq_swap(q, p, min_index);
        bubble_down(q, min_index);
    }
}
```

Для достижения позиции листа требуется $\lfloor \lg n \rfloor$ исполнений процедуры `bubble_down`, каждое из которых исполняется за линейное время. Таким образом, удаление корня занимает время $O(\log n)$.

Многократный обмен местами наибольшего элемента с последним элементом и вызов процедуры восстановления пирамиды дает нам алгоритм пирамидальной сортировки с временем исполнения $O(n \log n)$ (листинг 4.6).

**Листинг 4.6. Алгоритм пирамидальной сортировки**

```
void heapsort_(item_type s[], int n) {
    int i; /* Счетчик */
    priority_queue q; /* Память для пирамидальной сортировки */

    make_heap(&q, s, n);

    for (i = 0; i < n; i++) {
        s[i] = extract_min(&q);
    }
}
```

Пирамидальная сортировка является замечательным алгоритмом. Его легко реализовать, что подтверждает полный код, представленный в листингах 4.4–4.6. Время исполнения этого алгоритма в наихудшем случае равно $O(n \log n)$, что является наилучшим временем исполнения, которое можно ожидать от любого алгоритма сортировки. Сортировка выполняется «на месте», а это означает, что используется только память, содержащая массив с сортируемыми элементами. Впрочем, надо отметить, что в нашей реализации программы пирамидальная сортировка не выполняется «на месте», т. к. очередь с приоритетами создается в $q$, а не в $s$. Но каждый новый извлеченный элемент идеально помещается в ячейку, освобожденную сокращающейся пирамидой, оставляя за собой отсортированный массив. Хотя на практике другие алгоритмы сортировки могут оказаться немного быстрее, вы не ошибетесь, если предпочтете этот для сортировки данных в оперативной памяти компьютера.

Очереди с приоритетами являются очень полезными структурами данных. Вспомните, как мы их использовали на практике (см. *разд. 3.6*). Кроме этого, в *разд. 15.2* дается полный набор реализаций очереди с приоритетами.

## 4.3.4. Быстрый способ создания пирамиды (*)

Как мы видели, пирамиду из $n$ элементов можно создать за время $O(n \log n)$ методом пошаговой вставки элементов. Удивительно, но пирамиду можно создать еще быстрее, используя нашу процедуру `bubble_down` (см. листинг 4.5) и выполнив некоторый анализ.

Допустим, что мы упакуем $n$ ключей, предназначенных для построения пирамиды, в первые $n$ элементов массива очереди с приоритетами. Форма пирамиды будет правильной, но иерархия доминирования окажется полностью нарушена. Как можно исправить это положение?

Рассмотрим массив в обратном порядке, начиная с последней ($n$-й) ячейки. Эта ячейка является листом дерева и поэтому доминирует над своими несуществующими потомками. То же самое верно и для последних $n/2$ ячеек массива, поскольку все они являются листьями. Продолжая двигаться по массиву в обратном направлении, мы, в конце концов, дойдем до внутреннего узла, имеющего потомков. Этот элемент может не доминировать над своими потомками, но эти потомки представляют правильно построенные (хотя и небольшого размера) пирамиды.

Это как раз та ситуация, для исправления которой предназначена процедура `bubble_down` — восстановление правильности иерархии произвольного элемента пирамиды, находящегося сверху двух меньших пирамид. Таким образом мы можем создать пирамиду, выполнив $n/2$ вызовов процедуры `bubble_down`, как показано в листинге 4.7.

**Листинг 4.7. Алгоритм быстрого создания пирамиды**

```
void make_heap_fast(priority_queue *q, item_type s[], int n) {
    int i; /* Счетчик */

    q->n = n;
    for (i = 0; i < n; i++) {
        q->q[i + 1] = s[i];
    }

    for (i = q->n/2; i >= 1; i--) {
        bubble_down(q, i);
    }
}
```

Умножив количество вызовов ($n$) процедуры `bubble_down` на верхний предел сложности каждой операции ($O(\log n)$), мы получим время исполнения $O(n \log n)$. Однако это ничуть не быстрее, чем время исполнения алгоритма пошаговой вставки, описанного ранее.

Но обратите внимание, что это *действительно* верхний предел, т. к. фактически только последняя вставка требует $\lfloor \lg n \rfloor$ шагов. Вспомните, что время исполнения процедуры `bubble_down` пропорционально высоте пирамид, слияние которых она осуществляет. Большинство из этих пирамид очень небольшого размера. В полном двоичном дереве из $n$ узлов $n/2$ узлов являются листьями (т. е. имеют высоту 0), $n/4$ узлов имеют высоту 1, $n/8$ узлов имеют высоту 2 и т. д.). В общем, имеется самое большее $\lceil n/2^{h+1} \rceil$ узлов высотой $h$, соответственно затраты на создание пирамиды составляют:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \, h \le n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \le 2n.$$

Так как эта сумма, строго говоря, не является геометрической прогрессией, мы не можем выполнить обычную операцию тождества. Но можно быть уверенными, что несущественный вклад числителя ($h$) полностью перекроется значением знаменателя ($2^h$). Таким образом, функция быстро приближается к линейной.

Имеет ли значение тот факт, что мы можем создать пирамиду за линейное время вместо времени $O(n \log n)$? Не совсем. Время создания не доминирует над сложностью пирамидальной сортировки, поэтому улучшение времени создания не улучшает его производительность в наихудшем случае. Тем не менее перед нами убедительная демонстрация важности внимательного анализа и возможности получения дополнительных бонусов от сходимости геометрической прогрессии.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Расположение элемента в пирамиде

**ЗАДАЧА**. Для имеющейся пирамиды на основе массива из $n$ элементов и действительного числа $x$ разработайте эффективный метод определения, является ли $k$-й элемент пирамиды большим или равным $x$. Независимо от размера пирамиды время исполнения алгоритма в наихудшем случае должно быть равным $O(k)$. Подсказка: находить $k$-й наименьший элемент не нужно — требуется только определить его взаимосвязь с $x$.

**РЕШЕНИЕ**. Существуют по крайней мере два разных подхода, дающих правильные, но неэффективные алгоритмы для решения этой задачи.

1. Процедура выборки наименьшего значения вызывается $k$ раз, и каждое из выбранных значений проверяется, меньше ли оно, чем значение $x$. Таким образом явно сортируются первые $k$ элементов, что предоставляет нам больше информации, чем требует постановка задачи, но для этого требуется время $O(k \log n)$.
2. Наименьший $k$-й элемент не может находиться глубже, чем на $k$-м уровне пирамиды, поскольку путь от него до корня должен проходить через элементы с убывающими значениями. Таким образом, мы можем просмотреть все элементы первых $k$ уровней пирамиды и подсчитать, сколько из них меньше, чем $x$. Просмотр прекращается, когда мы либо нашли $k$ таких элементов, либо исчерпали все элементы. И хотя этот алгоритм дает правильное решение задачи, он исполняется за время $O(\min(n, 2^k))$, поскольку $k$ верхних уровней насчитывают $2^k - 1$ элементов.

Решение с временем исполнения $O(k)$ должно проверить только $k$ элементов меньших, чем $x$, плюс не более $O(k)$ элементов больших, чем $x$. Такое решение в виде рекурсивной процедуры, вызывающейся для корневого элемента с параметрами $i = 1$ и $count = k$, приводится в листинге 4.8.

**Листинг 4.8. Сравнение $k$-го элемента с числом $x$**

```
int heap_compare(priority_queue *q, int i, int count, int x) {
    if ((count <= 0) || (i > q->n)) {
        return(count);
    }

    if (q->q[i] < x) {
        count = heap_compare(q, pq_young_child(i), count-1, x);
```

```
        count = heap_compare(q, pq_young_child(i)+1, count, x);
    }

    return(count);
}
```

Если корневой элемент неубывающей бинарной пирамиды больше, чем $x$, тогда в оставшейся пирамиде не может быть элементов меньших, чем $x$, т. к. по определению этого типа пирамиды корневой элемент должен быть наименьшим. В противном случае процедура просматривает потомков всех узлов со значением меньшим, чем $x$, до тех пор, пока либо не найдет $k$ таких узлов (и в этом случае возвращается 0), либо не исчерпает все узлы (и тогда возвращается положительное значение). Таким образом, процедура найдет достаточное количество подходящих элементов при условии, что они имеются в пирамиде.

Но сколько времени потребуется? Процедура проверяет потомков только тех узлов, чье значение меньше, чем $x$, и общее количество таких узлов равно самое большее $k$. Для каждого из этих узлов проверяются самое большее два потомка, следовательно, количество проверяемых узлов равно самое большее $2k + 1$, а общее время исполнения равно $O(k)$. ■

## 4.3.5. Сортировка вставками

Теперь рассмотрим другой подход к сортировке, опирающийся на использование эффективных структур данных. Выбираем следующий элемент из неотсортированного списка и вставляем его в должную позицию в отсортированном списке. Псевдокод этого метода сортировки приведен в листинге 4.9.

**Листинг 4.9. Сортировка вставками**

```
for (i = 1; i < n; i++) {
    j = i;
    while ((j > 0) && (s[j] < s[j - 1])) {
        swap(&s[j], &s[j - 1]);
        j = j-1;
    }
}
```

Хотя время исполнения алгоритма сортировки вставками в наихудшем случае равно $O(n)$, его производительность значительно выше, если данные почти отсортированные, т. к. для помещения элемента в должное место достаточно лишь небольшого числа итераций внутреннего цикла.

Сортировка вставками является, возможно, самым простым примером метода сортировки *поэтапными вставками*, где мы создаем сложную структуру из $n$ элементов, сначала создав ее из $n - 1$ элементов, после чего осуществляем необходимые изменения, чтобы добавить последний элемент.

Обратите внимание, что быстрые алгоритмы сортировки на основе поэтапных вставок обеспечены эффективными структурами данных. Операция вставки в сбалансирован-

ное дерево поиска занимает время $O(\log n)$, а общее время создания дерева равно $O(n \log n)$. При симметричном обходе такого дерева элементы считываются в отсортированном порядке, что занимает линейное время.

## 4.4. История из жизни. Билет на самолет

Я взялся за эту работу в поисках справедливости. Меня наняла одна туристическая фирма, чтобы помочь им разработать алгоритм поиска самого дешевого маршрута для перелета из города $x$ в город $y$. Сумасшедшие колебания цен на авиабилеты, устанавливаемые согласно современным методам «управления доходами», повергали меня в полное недоумение. Возникало впечатление, что цены на авиарейсы взлетают намного эффективнее, чем сами самолеты. Мне казалось, проблема в том, что авиакомпании скрывают действительно низкие цены на их рейсы. Я рассчитывал, что если я справлюсь с поставленной задачей, это позволит мне в будущем покупать билеты подешевле.

— Смотрите, — начал я свою речь на первом совещании. — Все не так уж сложно. Создаем граф, в котором вершины представляют аэропорты, и соединяем ребром каждую пару аэропортов $u$ и $v$, между которыми есть прямой рейс. Устанавливаем вес этого ребра равным стоимости самого дешевого имеющегося в наличии билета из $u$ в $v$. Теперь самая низкая цена перелета из города $x$ в город $y$ будет соответствовать самому короткому пути $x - y$ между соответствующими точками графа. Этот путь можно найти с помощью алгоритма Дейкстры для поиска кратчайшего пути. Проблема решена! — провозгласил я, эффектно взмахнув руками.

Члены собрания задумчиво покивали головами, потом разразились смехом. Мне предстояло кое-что узнать о чрезвычайно сложном процессе формирования цен на билеты пассажирских авиарейсов. В любой момент времени существует буквально миллион разных цен, при этом в течение дня они меняются несколько раз. Цена определяется сложным набором правил, которые являются общими для всей отрасли пассажирских авиаперевозок и представляют собой запутанную систему, не основанную на каких-либо логических принципах. Исключением из общего правила являлась только восточноафриканская страна Малави. Обладая населением в 18 миллионов человек и доходом на душу населения в 1234 доллара (180-е место в мире), она неожиданно оказалась серьезным фактором, формирующим политику ценообразования мировых пассажирских авиаперевозок. Чтобы определить точную стоимость любого авиамаршрута, требовалось убедиться, что он не проходит через Малави.

Настоящая сложность состояла в том, что для первого отрезка маршрута — скажем, от Лос-Анджелеса (аэропорт LAX) до Чикаго (аэропорт ORD), существует сотня разных тарифов, и не меньшее количество тарифов может существовать для каждого последующего отрезка маршрута — например, от Чикаго до Нью-Йорка (аэропорт JFK). Самый дешевый билет для отрезка LAX–ORD (скажем, для детей членов Американской ассоциации пенсионеров) может быть несовместим с самым дешевым билетом для отрезка ORD–JFK (если, например, он является специальным предложением для мусульман, которое можно использовать только с последующей пересадкой на рейс в Мекку).

Признав справедливость критики за чрезмерное упрощение задачи, я принялся за работу. И начал со сведения задачи к наиболее простому случаю.

— Хорошо. Скажем, вам нужно найти самый дешевый билет для рейса с одной пересадкой (т. е. двухэтапного), который проходит вашу проверку соответствия правилам. Есть ли какой-либо способ заранее определить, какие пары этапов маршрута пройдут проверку, и не выполнять при этом саму проверку?

— Нет никакой возможности знать это наперед, — заверили меня. — Мы только можем выполнить процедуру «черного ящика», чтобы решить, существует ли конкретная цена на билет для конкретного маршрута и для конкретного пассажира.

— Значит, наша цель заключается в том, чтобы вызывать эту процедуру для минимального количества комбинаций билетов. Это означает, что нужно оценивать все возможные комбинации билетов, начиная с самой дешевой до самой дорогой, до тех пор, пока мы не найдем первую удовлетворяющую правилам комбинацию.

— Совершенно верно.

— Тогда почему бы нам не составить набор всех возможных $m \times n$ пар билетов, отсортировать их по стоимости, после чего оценить их в отсортированной последовательности? Безусловно, это можно сделать за время $O(nm \log(nm))$[4].

— Это мы сейчас и делаем, но составление полного набора $m \times n$ пар обходится довольно дорого, притом что подходящей может оказаться первая пара.

Я понял, что задача действительно интересная.

— Что вам действительно нужно, так это эффективная структура данных, которая многократно возвращает *следующую* наиболее дорогую пару билетов, при этом не составляя все пары наперед.

Это в самом деле было интересно. Поиск наибольшего элемента в наборе значений, подвергающемся вставкам и удалениям, является *как раз той задачей*, для решения которой замечательно подходят очереди с приоритетами. Но проблема в рассматриваемом случае заключалась в том, что мы не могли заранее заполнить ключами очередь с приоритетами. Новые пары нужно было вставлять в очередь после каждой оценки.

Я составил несколько примеров, как показано на рис. 4.3.

Каждую возможную цену билета двухэтапного рейса можно было представлять списком индексов ее компонентов (т. е. цен билетов каждого отрезка маршрута). Безусловно, самый дешевый билет для всего маршрута будет получен сложением самых дешевых билетов для каждого отрезка маршрута. В нашем представлении это будет билет (1, 1). Следующий самый дешевый билет получается сложением первого билета одного отрезка маршрута и второго билета другого маршрута, т. е. это будет или (1, 2) или (2, 1). Дальше становится сложнее. Третьим самым дешевым билетом могла бы быть неиспользованная пара из приведенных двух или же пара (1, 3) или (3, 1). И действительно, это могла бы быть пара (3, 1), если бы третья по величине цена билета на отрезок $X$ маршрута была 120 долларов.

---

[4] Вопрос, можно ли отсортировать все такие суммы быстрее, чем $nm$ произвольных целых чисел, представляет собой нерешенную задачу теории алгоритмов. Дополнительную информацию по сортировке $X + Y$ (так называется эта задача) можно найти в книгах [Fre76] и [Lam92].

| X | Y |
|---|---|
| $100 | $50 |
| $110 | $75 |
| $130 | $125 |

| X+Y |
|---|
| $150 (1,1) |
| $160 (2,1) |
| $175 (1,2) |
| $180 (3,1) |
| $185 (2,2) |
| $205 (3,2) |
| $225 (1,3) |
| $235 (2,3) |
| $255 (3,3) |

**Рис. 4.3.** Сортировка в возрастающем порядке сумм $X$ и $Y$

— Скажите, — спросил я, — у нас есть время, чтобы отсортировать эти оба списка цен билетов в возрастающем порядке?

— В этом нет надобности, — ответил руководитель группы. — Они подаются из базы данных в отсортированном порядке.

Это была хорошая новость. Нам не нужно было исследовать пары цен $(i + 1, j)$ и $(i, j + 1)$ до рассмотрения пары $(i, j)$, поскольку очевидно, что они были более высокими.

— Есть, — сказал я. — Мы будем отслеживать пары индексов в очереди с приоритетами, а ключом пары будет сумма цен билетов. Вначале в очередь вставляется только одна пара цен: (1, 1). Если эта пара оказывается неподходящей, мы вставляем в очередь две следующие пары: (1, 2) и (2, 1). В общем, после исследования и выбраковки пары $(i, j)$ мы последовательно вставляем в очередь пары $(i + 1, j)$ и $(i, j + 1)$. Таким образом, мы исследуем все пары в правильном порядке.

Компания быстро уловила суть решения.

— Конечно. Но как насчет дубликатов? Мы же будем создавать пару $(x, y)$ двумя разными способами: при расширении пар $(x - 1, y)$ и $(x, y - 1)$.

— Вы правы. Нам нужна дополнительная структура данных, чтобы предотвратить появление дубликатов. Самым простым решением будет использование хеш-таблицы для проверки существования той или иной пары в очереди с приоритетами до ее вставки. В действительности, структура данных никогда не будет содержать больше чем $n$ активных пар, поскольку с каждым отдельным значением цены первого отрезка маршрута можно создать только одну пару цен.

Решение было принято. Наш подход естественным образом обобщался для маршрутов с более чем двумя отрезками (при этом сложность возрастала с увеличением количества этапов маршрута). Метод выбора первого оптимального варианта, свойственный нашей очереди с приоритетами, позволил системе прекращать поиск, как только она находила действительно самый дешевый билет. Такой подход оказался достаточно быстрым, чтобы предоставить интерактивный ответ пользователю. Однако я не заметил, чтобы в результате этого авиабилеты хоть немного подешевели.

# 4.5. Сортировка слиянием. Метод «разделяй и властвуй»

Рекурсивные алгоритмы разбивают большую задачу на несколько подзадач. При рекурсивном подходе сортируемые элементы разбиваются на две группы, каждая из этих меньших групп сортируется рекурсивно, после чего оба отсортированных списка сливаются воедино и их элементы чередуются, образуя полностью отсортированный общий список. Этот алгоритм называется *сортировкой слиянием* (mergesort):

```
Mergesort(A[1, ..., n])
   Merge( MergeSort(A[1, ..., _n/2]), MergeSort(A[_n/2 + 1, ..., n]) )
```

Базовый случай рекурсивной сортировки имеет место, когда исходный массив состоит самое большее из одного элемента, вследствие чего в нем не требуется никаких перестановок. На рис. 4.4 показана графическая иллюстрация работы алгоритма сортировки слиянием. Сортировку слиянием можно представлять себе как симметричный обход дерева, когда каждое слияние происходит после возврата упорядоченных подмассивов вызовами к двум потомкам.

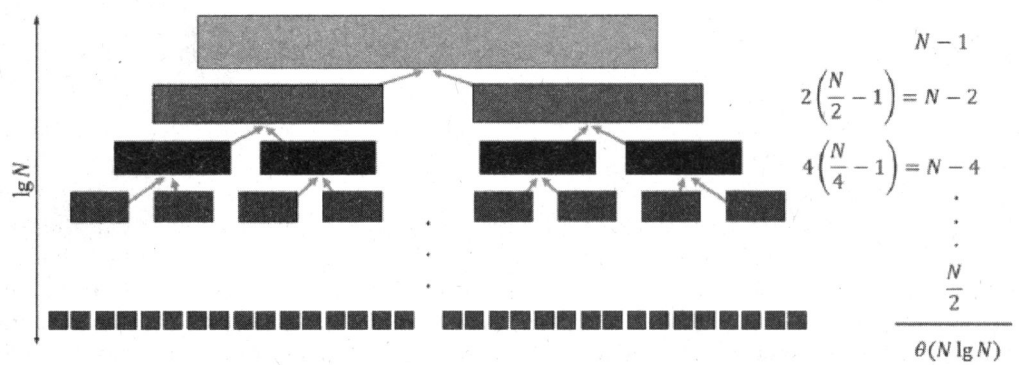



**Рис. 4.4.** Рекурсивное дерево для алгоритма сортировки слиянием. Высота дерева составляет $\lceil \lg_2 n \rceil$, а стоимость операций слияний на каждом уровне равна $\Theta(n)$, что дает нам алгоритм с временной сложностью, равной $\Theta(n \log n)$

Эффективность сортировки слиянием зависит от эффективности слияния двух отсортированных частей в один отсортированный список. Одним из способов было бы объединить обе части и выполнить сортировку получившегося списка методом пирамидальной сортировки или каким-либо другим способом, но это свело бы на нет все наши усилия, потраченные на сортировку этих частей по отдельности.

Поэтому списки *сливаются* в один следующим образом. Оба списка отсортированы в возрастающем порядке, значит, наименьший элемент должен находиться в самом начале одного из них. Этот наименьший элемент перемещается в общий список, при этом один из списков укорачивается на один элемент. Следующий наименьший элемент теперь должен быть самым первым в одном из оставшихся двух списков. Повторяя эту операцию до тех пор, пока не опустеют оба отсортированных списка, мы сливаем эти два списка (с общим количеством элементов $n$) в один отсортированный список, выполняя самое большее $n - 1$ сравнений за время $O(n)$.

Какое же общее время исполнения сортировки слиянием? Чтобы ответить на этот вопрос, следует принять во внимание, какой объем работы выполняется на каждом уровне дерева сортировки, как показано на рис. 4.4. Если мы допустим для простоты, что $n$ является степенью двойки, то $k$-й уровень содержит все $2^k$ вызовов процедуры сортировки слиянием `mergesort`, обрабатывающих поддиапазоны из $n/2^k$ элементов.

Работа, выполняемая на нулевом (верхнем) уровне, заключается в слиянии двух отсортированных списков, каждый размером $n/2$, и при этом происходит самое большее $n-1$ сравнений. Работа, выполняемая на первом уровне (на уровень ниже), состоит в слиянии двух отсортированных списков, каждый размером $n/4$, и при этом происходит самое большее $n-2$ сравнений. В общем, работа, выполняемая на $k$-м уровне, состоит в слиянии $2^k$ пар отсортированных списков, каждый размером $n/2^{k+1}$, и при этом происходит самое большее $n-2^k$ сравнений. *Слияние всех элементов на каждом уровне выполняется за линейное время.* Каждый из $n$ элементов фигурирует только в одной подзадаче на каждом уровне. Наиболее трудоемким случаем (по количеству сравнений) является самый верхний уровень.

На каждом уровне количество элементов в подзадаче делится пополам. Количество делений пополам числа $n$, производимых до тех пор, пока не будет получена единица, равно $\lceil \lg_2 n \rceil$. Так как глубина рекурсии составляет $\lg n$ уровней, а каждый уровень обрабатывается за линейное время, то в наихудшем случае время исполнения алгоритма сортировки слиянием равно $O(n \log n)$.

Алгоритм сортировки слиянием хорошо подходит для сортировки связных списков, поскольку он, в отличие от алгоритмов пирамидальной и быстрой сортировки, не основывается на произвольном доступе к элементам. Их главным недостатком является необходимость во вспомогательном буфере при сортировке массивов. Отсортированные связные списки можно легко слить вместе, не требуя дополнительной памяти, а просто упорядочивая указатели. Но для слияния двух отсортированных массивов (или частей массива) требуется третий массив — для хранения результатов слияния, чтобы не потерять содержимое сливаемых массивов. Допустим, что мы сливаем множества {4, 5, 6} и {1, 2, 3}, записанные слева направо в одном массиве. Без использования буфера нам придется записывать отсортированные элементы поверх элементов левой половины массива, вследствие чего последние будут утеряны.

Сортировка слиянием является классическим примером алгоритмов типа «разделяй и властвуй». Мы всегда в выигрыше, когда можем разбить одну большую задачу на две подзадачи, т. к. меньшие задачи легче решить. Секрет заключается в том, чтобы воспользоваться двумя частичными решениями для создания решения всей задачи, как это было сделано с операцией слияния. Парадигма «разделяй и властвуй» играет важную роль в разработке алгоритмов и рассматривается в *главе 5*.

### Реализация

Код процедуры алгоритма сортировки слиянием представлен в листинге 4.10.

**Листинг 4.10. Код реализации алгоритма сортировки слиянием**

```
void merge_sort(item_type s[], int low, int high) {
    int middle; /* Индекс среднего элемента */
```

```
    if (low < high) {
        middle = (low + high) / 2;
        merge_sort(s, low, middle);
        merge_sort(s, middle + 1, high);

        merge(s, low, middle, high);
    }
}
```

Но реализация части алгоритма, в которой осуществляется слияние отсортированных списков, оказывается более сложной. Проблема состоит в том, что нам нужно где-то хранить слитый массив. Чтобы избежать потери элементов в процессе слияния, мы сначала создаем копии исходных массивов (листинг 4.11).

**Листинг 4.11. Процедура слияния массивов**

```
void merge(item_type s[], int low, int middle, int high) {
    int i; /* Счетчик */
    queue buffer1, buffer2; /* Буфер для хранения объединяемых элементов */

    init_queue(&buffer1);
    init_queue(&buffer2);

    for (i = low; i <= middle; i++) enqueue(&buffer1, s[i]);
    for (i = middle + 1; i <= high; i++) enqueue(&buffer2, s[i]);

    i = low;
    while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
        if (headq(&buffer1) <= headq(&buffer2)) {
            s[i++] = dequeue(&buffer1);
        } else {
            s[i++] = dequeue(&buffer2);
        }
    }

    while (!empty_queue(&buffer1)) {
        s[i++] = dequeue(&buffer1);
    }

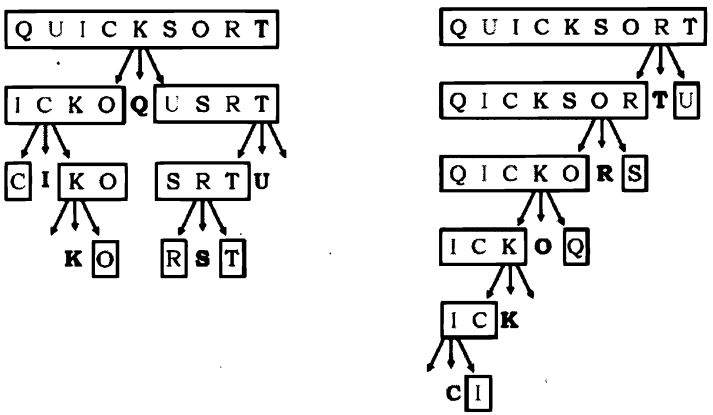
    while (!empty_queue(&buffer2)) {
        s[i++] = dequeue(&buffer2);
    }
}
```

# 4.6. Быстрая сортировка. Рандомизированная версия

Алгоритм *быстрой сортировки* (quicksort) работает следующим образом. Из массива размером $n$ выбирается произвольный элемент $p$. Оставшиеся $n - 1$ элементов массива разделяются на две части: левую (или нижнюю), содержащую все элементы, меньшие

элемента $p$, и правую (или верхнюю), содержащую все элементы, большие $p$. Элемент $p$ занимает отдельную ячейку между этими двумя частями.

Графическая иллюстрация работы алгоритма быстрой сортировки представлена на рис. 4.5.




*а* *б*

**Рис. 4.5.** Графическая иллюстрация работы алгоритма быстрой сортировки: сначала в каждом подмассиве в качестве разделителя выбирается первый элемент (*а*), а затем последний (*б*)

Такое разбиение массива на две части преследует две цели. Во-первых, опорный элемент $p$ находится в точно той же позиции массива, в которой он будет находиться в конечном отсортированном массиве. Во-вторых, после разделения массива на части в конечной отсортированной последовательности элементы не перемещаются из одной части в другую. *Таким образом, сортировку элементов в левой и правой частях массива можно выполнять независимо друг от друга.* Это дает нам рекурсивный алгоритм сортировки, поскольку мы можем использовать подход с разбиением массива на две половины для сортировки каждой первоначальной половины. Такой алгоритм должен быть правильным, поскольку, в конечном счете, каждый элемент оказывается в правильном месте. В листинге 4.12 приведен код алгоритма на языке C.

**Листинг 4.12. Код алгоритма быстрой сортировки**

```
void quicksort(item_type s[], int l, int h) {
int p; /* Индекс элемента-разделителя */

if (l < h) {
p = partition(s, l, h);
quicksort(s, l, p - 1);
quicksort(s, p + 1, h);
}
}
```

Массив можно разбить на части за один проход с линейным временем исполнения для определенного опорного элемента посредством постоянного сопровождения трех час-

тей массива: содержащей элементы меньшие, чем опорный элемент (слева от firsthigh), содержащей элементы равные или большие, чем опорный (между firsthigh и i) и содержащей непроверенные элементы (справа от i). Реализация процедуры разбиения приведена в листинге 4.13.

**Листинг 4.13. Процедура разбиения массива на части**

```
int partition(item_type s[], int l, int h) {
    int i;          /* Счетчик */
    int p;          /* Индекс элемента-разделителя */
    int firsthigh;  /* Позиция для элемента разделителя */

    p = h;      /* Выбираем последний элемент в качестве разделителя */
    firsthigh = l;
    for (i = l; i < h; i++) {
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    }
    swap(&s[p], &s[firsthigh]);

    return(firsthigh);
}
```

Так как процедура разделения содержит самое большее $n$ операций обмена местами двух элементов, то разбиение массива выполняется за линейное время относительно количества ключей. Но каково общее время исполнения алгоритма быстрой сортировки? Подобно алгоритму сортировки слиянием алгоритм быстрой сортировки создает рекурсивное дерево вложенных поддеревьев массива из $n$ элементов. Подобно алгоритму сортировки слиянием алгоритм быстрой сортировки обрабатывает (не слиянием частей в один массив, а, наоборот, разбиением массива на части) элементы каждого подмассива на каждом уровне за линейное время. Опять же, подобно алгоритму сортировки слиянием общее время исполнения алгоритма быстрой сортировки равно $O(n \cdot h)$, где $h$ — высота рекурсивного дерева.

Но трудность здесь состоит в том, что высота дерева зависит от конечного местонахождения опорного элемента в каждой части массива. Если нам очень повезет и опорным каждый раз будет элемент, находящийся посередине массива, то размер полученных вследствие такого деления подзадач всегда будет равен половине размера задачи предыдущего уровня. Высота представляет количество делений, которым подвергается массив и последующие подмассивы до тех пор, пока полученный подмассив не будет состоять из одной ячейки, что соответствует $h = \lceil \lg n \rceil$. Такая благоприятная ситуация показана на рис. ЦВ-4.6, *а* и представляет наилучший случай алгоритма быстрой сортировки.

Теперь допустим, что нам постоянно не везет и что наш выбор опорного элемента все время разбивает массив самым неравномерным образом. Это означает, что в качестве опорного всегда выбирается наибольший или наименьший элемент текущего массива.

После установки этого опорного элемента в должную позицию у нас остается одна подзадача размером $n - 1$ элементов. То есть мы тратим линейное время на ничтожно малое уменьшение задачи — всего лишь на один элемент (рис. ЦВ-4.6, *б*). Чтобы разбить массив так, чтобы на каждом уровне находился хотя бы один элемент, требуется дерево высотой $n - 1$, а время выполнения в наихудшем случае будет $\Theta(n^2)$.

Таким образом, время исполнения алгоритма быстрой сортировки в наихудшем случае хуже, чем для пирамидальной сортировки или сортировки слиянием. Чтобы оправдать свое название, алгоритму быстрой сортировки следовало бы работать лучше в среднем случае. Для понимания этого требуется почувствовать произвольную выборку на интуитивном уровне.

## 4.6.1. Интуиция: ожидаемое время исполнения алгоритма быстрой сортировки

Ожидаемое время исполнения алгоритма быстрой сортировки зависит от высоты дерева разбиения первоначального массива, создаваемого опорными элементами на каждом шаге разбиения. Мы рекурсивно разбиваем общее количество элементов на две равные части, после чего сливаем в требуемом порядке за линейное время. Поэтому время исполнения алгоритма сортировки слиянием равно $O(n \log n)$. Таким образом, всякий раз, когда опорный элемент находится возле центра сортируемого массива (т. е. разделение проходит возле среднего элемента), мы получаем такое же хорошее разбиение и такое же время исполнения, как и для алгоритма сортировки слиянием.

Я дам не строгое, а опирающееся на интуицию объяснение, почему время исполнения алгоритма быстрой сортировки в среднем случае равно $O(n \log n)$. Какова вероятность того, что выбранный в произвольном порядке разделяющий элемент окажется хорошим? Самым лучшим разделяющим элементом был бы средний элемент массива, поскольку по каждую его сторону оказалось бы ровно по половине элементов исходного массива. Но вероятность выбрать наудачу точно средний элемент довольно-таки низка, а именно равна $1/n$.

Но допустим, что разделитель является *достаточно хорошим*, если он находится в центральной половине массива, т. е. в диапазоне элементов для сортировки от $n/4$ до $3n/4$. Таких достаточно хороших разделительных элементов имеется довольно много, поскольку половина всех элементов расположена ближе к центру массива, чем к его краям (рис. 4.7). Таким образом, вероятность выбора достаточно хорошего разделительного элемента при каждом выборе равна 1/2. Выбор хороших разделительных элементов будет способствовать повышению эффективности сортировки.

При выборе самого худшего возможного *достаточно хорошего* разделителя большая часть разделенного массива содержит $3n/4$ элемента. Это также оказывается ожидае-

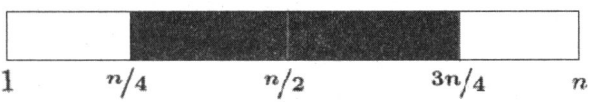



**Рис. 4.7.** В половине случаев разделительный элемент расположен ближе к середине массива, чем к его краям

мым размером большей секции, которая остается после выбора произвольного опорного элемента $p$ по медиане между наихудшим возможным опорным элементом ($p = 1$ или $p = n$, дающим секцию размером $n - 1$) и наилучшим возможным опорным элементом ($p = n/2$, дающим две секции размером $n/2$). Так какой же будет высота $h_g$ дерева алгоритма быстрой сортировки, созданного на основе ожидаемого значения разбиения? Самый длинный путь по этому дереву проходит через части размером $n$, $(3/4)n$, $(3/4)^2n$, ..., и т. д. до 1 элемента. Сколько раз можно умножить $n$ на 3/4, пока мы не дойдем до 1? Поскольку

$$(3/4)^{h_g} n = 1 \Rightarrow n = (4/3)^{h_g}, \text{ то } h_g = \log_{4/3} n.$$

В среднем деревья разделов алгоритма быстрой сортировки, созданные посредством произвольного выбора (и аналогично двоичные деревья поиска, созданные произвольными вставками), дают очень хорошие результаты. Более подробный анализ показывает, что после $n$ вставок средняя высота дерева составляет приблизительно $2 \ln n$. Так как $2 \ln n \approx 1{,}386 \lg n$, то это всего лишь на 39% выше, чем высота идеально сбалансированного двоичного дерева. Поскольку время обработки каждого уровня составляет $O(n)$, то среднее время исполнения алгоритма быстрой сортировки равно $O(n \log n)$. Если нам *особенно* не повезет и наши произвольно выбранные разделители всегда будут оказываться наибольшими или наименьшими элементами массива, то алгоритм быстрой сортировки превратится в сортировку методом выбора с временем исполнения $O(n^2)$. Но вероятность такого развития событий чрезвычайно мала.

## 4.6.2. Рандомизированные алгоритмы

Следует отметить одну важную тонкость в ожидаемом времени исполнения $O(n \log n)$ алгоритма быстрой сортировки. В нашей реализации алгоритма в предыдущем разделе мы выбирали в качестве разделителя последний элемент каждого подмассива (части предыдущего массива). Допустим, мы используем этот алгоритм на отсортированном массиве. В таком случае при каждом разбиении будет выбираться наихудший из всех возможных элемент-разделитель, а время исполнения станет квадратичным. Для любого детерминистского способа выбора элемента-разделителя существует наихудший входной экземпляр с квадратичным временем исполнения. В представленном ранее анализе утверждается лишь следующее:

> существует высокая вероятность, что время исполнения алгоритма быстрой сортировки будет равно $\Theta(n \log n)$ — *при условии*, что подлежащие сортировке данные идут в произвольном порядке.

Теперь допустим, что прежде чем приступать к сортировке $n$ элементов, мы переупорядочим их произвольным образом. Эту операцию перестановки можно выполнить за время $O(n)$ (подробно об этом рассказано в *разд. 16.7*). Эта кажущаяся расточительность гарантирует ожидаемое время исполнения $O(n \log n)$ для *любого* входного экземпляра задачи. Хотя наихудший случай времени исполнения продолжает оставаться возможным, теперь он зависит исключительно от того, насколько нам повезет или не повезет. Но явно определенного наихудшего входного экземпляра больше нет. Следовательно, теперь мы можем сказать:

существует высокая вероятность, что время исполнения рандомизированного алгоритма быстрой сортировки будет равно $\Theta(n \log n)$ для *любого* входного экземпляра задачи.

В качестве альтернативного варианта мы могли бы прийти к такому же утверждению, выбирая произвольный элемент-разделитель на каждом шаге.

*Рандомизация* является мощным инструментом для улучшения алгоритмов с плохой временной сложностью в наихудшем случае, но с хорошей сложностью в среднем случае. С ее помощью алгоритмы можно сделать более устойчивыми в граничных случаях и более эффективными на высокоструктурированных вводных экземплярах, которые делают неэффективными эвристические механизмы принятия решений (как в случае с отсортированным входом для алгоритма быстрой сортировки). Рандомизацию часто можно применять с простыми алгоритмами, обеспечивая таким образом ожидаемую производительность, которую в противном случае можно получить, только используя сложные детерминистские алгоритмы. Рандомизированные алгоритмы рассматриваются в *главе 6*.

Чтобы должным образом анализировать рандомизированные алгоритмы, необходимо иметь определенные познания в области теории вероятностей. Краткий обзор этой темы приводится в *главе 6* с последующим более подробным рассмотрением рандомизированных алгоритмов. Но некоторые подходы к разработке эффективных рандомизированных алгоритмов можно с легкостью объяснить, что мы и сделаем здесь.

- *Рандомизированная выборка.*

  Допустим, мы хотим получить общее представление о среднем значении $n$ элементов, но у нас нет ни достаточного времени, ни места, чтобы просмотреть все значения. В таком случае мы делаем выборку произвольных элементов из всего множества и определяем медиану этих элементов. Полученный результат должен быть репрезентативным для всего множества.

  Эта идея лежит в основе исследований путем опроса, когда берется выборка мнений небольшого количества людей для представления мнения всего населения. Но если не обеспечить действительно *произвольную* выборку, а опросить $x$ первых встречных, то возможны искажения в ту или иную сторону. Во избежание таких искажений выполняющие опрос агентства обычно звонят по произвольным телефонным номерам и надеются, что кто-либо ответит.

- *Рандомизированное хеширование.*

  Мы уже говорили, что посредством хеширования можно реализовать словарный поиск с ожидаемым временем исполнения $O(1)$. Но для любой хеш-функции имеется наихудший случай в виде набора ключей, которые хешируются в одну корзину. Однако допустим, что первым шагом алгоритма мы выбираем произвольную функцию хеширования из большого семейства подходящих функций. Тогда мы получаем такую же улучшенную гарантию, как и для рандомизированного алгоритма быстрой сортировки.

- *Рандомизированный поиск.*

  Рандомизацию можно также применять для организации методов поиска, таких как имитация отжига (simulated annealing). Подробно это обсуждается в *разд. 12.6.3*.

## ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Болты и гайки

**ЗАДАЧА**. Задача *болтов и гаек* определяется таким образом. Есть набор разных $n$ гаек и такое же количество соответствующих болтов. Вам нужно найти наиболее эффективным способом для каждого болта подходящую гайку, пробуя, подходит ли выбранная гайка к тому или иному болту. Сравнивать можно только болты с гайками, т. к. разница между членами пары гаек или болтов слишком маленькая, чтобы ее можно было увидеть глазом, и поэтому нельзя сравнивать гайки с гайками или болты с болтами.

Разработайте алгоритм для решения этой задачи за время $O(n^2)$, а потом разработайте рандомизированный алгоритм для решения этой задачи за ожидаемое время $O(n \log n)$.

**РЕШЕНИЕ**. Алгоритм полного перебора решает эту задачу, сравнивая первый болт со всеми гайками, пока не найдет подходящую, после чего переходит к следующему болту и сравнивает его со всеми оставшимися гайками, и т. д. В худшем случае для первого болта потребуется $n$ сравнений. Повторение этой процедуры для всех последующих болтов со всеми остающимися гайками дает нам алгоритм с квадратичным числом сравнений.

Но что, если вместо последовательного перебора болтов, начиная с первого, мы каждый раз будем брать произвольный болт? В среднем мы можем ожидать перебора около половины гаек, пока не найдем подходящую для выбранного болта, поэтому этот рандомизированный алгоритм в среднем будет иметь ожидаемое время исполнения наполовину меньшее, чем в худшем случае. Это можно рассматривать как определенное улучшение, хотя и не асимптотического типа.

Поскольку рандомизированный алгоритм быстрой сортировки обеспечивает требуемое ожидаемое время исполнения, то будет естественной идея эмулировать его для решения этой задачи. Основной операцией алгоритма быстрой сортировки является разбиение массива элементов на две части по элементу-разделителю. Можем ли мы разбить множества гаек и болтов по произвольно выбранному болту $b$?

Определенно, что, сравнивая размер гаек с размером произвольно выбранного болта $b$, мы можем разбить множество гаек на две части: меньших и больших, чем гайка, подходящая для болта $b$. Но нам также нужно разбить на части множество болтов по этому же произвольно выбранному болту размером $b$, а мы не можем сравнивать болты друг с другом. Но ведь когда мы найдем гайку подходящего размера для болта размером $b$, то можем использовать ее для разбиения болтов, точно так же, как мы использовали болт для разбиения на части множества гаек. Разбиение на части гаек и болтов происходит за $2n - 2$ сравнения, а время исполнения оставшихся операций следует непосредственно из анализа рандомизированного алгоритма быстрой сортировки.

Эта задача интересна тем, что для нее не существует простого детерминистического алгоритма. Она хорошо демонстрирует, как использование рандомизации позволяет избавиться от плохих входных экземпляров задачи посредством простого и изящного алгоритма. ■

### 4.6.3. Действительно ли алгоритм быстрой сортировки работает быстро?

Существует четкое асимптотическое различие между алгоритмом с временем исполнения $\Theta(n \log n)$ и алгоритмом с временем исполнения $\Theta(n^2)$. Только самый недоверчивый читатель будет сомневаться в моем утверждении, что алгоритмы сортировки слиянием, пирамидальной сортировки и быстрой сортировки покажут лучшую производительность на достаточно больших входных экземплярах задачи, чем алгоритмы сортировки вставками или сортировки методом выбора.

Но как можно сравнить два алгоритма с временной сложностью $\Theta(n \log n)$, чтобы решить, который из них быстрее? Как можно доказать, что алгоритм быстрой сортировки действительно быстрый? К сожалению, модель RAM и асимптотический анализ являются слишком грубыми инструментами для сравнений такого типа. В случае алгоритмов с одинаковой асимптотической сложностью детали их реализации и особенности программной и аппаратной платформы, на которой они исполняются, такие как объем оперативной памяти и производительность кэша, часто оказываются решающим фактором.

Можно только отметить, что в процессе экспериментирования было установлено, что должным образом реализованный алгоритм быстрой сортировки обычно в 2–3 раза быстрее, чем алгоритм сортировки слиянием или пирамидальной сортировки. Основной причиной этого является тот факт, что в алгоритме быстрой сортировки операции внутреннего цикла менее сложные. Но если вы не верите, что алгоритм быстрой сортировки быстрее, я не смогу вам этого доказать. Ответ на этот вопрос лежит за рамками применения аналитических инструментов, рассматриваемых в этой книге. Самым лучшим способом будет реализовать оба алгоритма и определить их эффективность экспериментальным способом.

## 4.7. Сортировка распределением. Метод блочной сортировки

Имена в телефонной книге можно отсортировать по первой букве фамилии. Таким образом у нас получится 26 разных корзин имен[5]. Обратите внимание, что любая фамилия в корзине *J* должна находиться после всех фамилий из корзины *I*, но перед любой фамилией из корзины *K*. Вследствие этого обстоятельства фамилии можно отсортировать в каждой отдельной корзине, после чего просто объединить отсортированные корзины.

Предполагая, что имена распределены среди корзин равномерно, каждая из получившихся 26 подзадач сортировки должна быть значительно меньшего размера, чем первоначальная задача. Далее, разделяя каждую корзину по *второй* букве фамилии, потом третьей и т. д., мы создаем корзины все меньшего и меньшего размера (т. е. содержащие все меньше и меньше элементов-фамилий). Список фамилий будет полностью отсортирован, когда вследствие таких делений каждая корзина станет содержать только

[5] В английском алфавите 26 букв. — *Прим. пер.*

одну фамилию. Только что описанный алгоритм сортировки называется *блочной сортировкой* (bucket sort) или *сортировкой распределением* (distribution sort).

Применение корзин является очень эффективным подходом, когда мы уверены, что данные распределены приблизительно равномерно. Эта же идея лежит в основе хеш-таблиц, *kd*-деревьев и многих других практических структур данных. Обратной стороной этой медали является то, что производительность может быть ужасной, если распределение данных окажется не таким, на какое мы рассчитывали. Хотя для структур данных типа сбалансированных двоичных деревьев гарантируется производительность худшего случая для входных данных с любым распределением, такая гарантия отсутствует для эвристических структур данных с неожиданным распределением ввода.

Неравномерное распределение данных встречается и в реальной жизни. Возьмем, например, такую необычную американскую фамилию, как Shiffiett. Когда я последний раз открывал телефонный справочник Манхэттена (в котором свыше миллиона фамилий), то в нем было пять человек с такой фамилией. Как вы думаете, сколько людей по фамилии Shiffiett проживает в небольшом городке с населением в 50 000? На рис. 4.8 показан фрагмент телефонного справочника города Шарлотсвилл в штате Вирджиния. Фамилия Shiffiett занимает в справочнике более *двух с половиной страниц*.

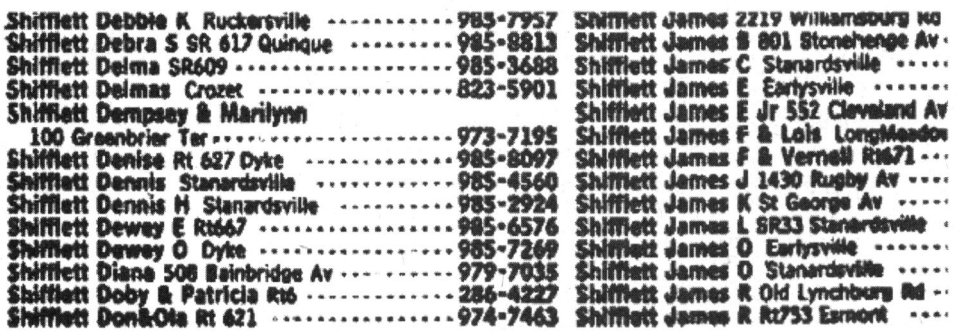



**Рис. 4.8.** Фрагмент телефонного справочника города Шарлотсвилл в штате Вирджиния

Клан Shiffiett присутствует в этом регионе много лет, и это обстоятельство огорчит любую программу сортировки распределением, поскольку при последовательном разбиении корзины *S* на *Sh*, на *Shi*, на *Shif*, на... на *Shifflett* в действительности не происходит никакого значительного разбиения.

#### *Подведение итогов*

Сортировку можно использовать для иллюстрации многих парадигм разработки алгоритмов. Методы структур данных, принцип «разделяй и властвуй», рандомизация и поэтапная обработка — все эти подходы позволяют разрабатывать эффективные алгоритмы сортировки.

## 4.7.1. Нижние пределы для сортировки

Обсудим последний вопрос, касающийся сложности алгоритмов сортировки. Мы знаем несколько алгоритмов сортировки. Для всех время исполнения в наихудшем случае

было равно $O(n \log n)$, но ни один из них не выполнялся за линейное время. Для сортировки $n$ элементов неизбежно требуется рассмотреть каждый из них, поэтому временная сложность любого алгоритма сортировки в наихудшем случае должна быть $\Omega(n)$. Возможно ли выполнить сортировку за линейное время?

К сожалению, предполагая, что используемый алгоритм основан на сравнении пар элементов, — нет. Нижнюю границу $\Omega(n \log n)$ можно установить на основе того обстоятельства, что любой алгоритм сортировки должен вести себя по-разному при сортировке каждой из возможных разных $n!$ перестановок $n$ элементов. Даже если бы алгоритм выполнял *абсолютно одинаковый* набор операций над двумя разными входными перестановками, ни при каких обстоятельствах обе они не были бы правильно упорядочены на выходе. Время исполнения любого алгоритма сортировки на основе сравнений определяется результатом каждого попарного сравнения. Набор всех возможных исполнений для такого алгоритма можно представить в виде дерева с $n!$ листьями, каждый из которых соответствует одной входной перестановке, а каждый путь от корня к концевому узлу описывает сравнения, выполненные для сортировки этого ввода. Минимальная высота дерева соответствует самому быстрому возможному алгоритму, и получается, что $\lg(n!) = \Theta(n \log n)$.

На рис. 4.9 показано дерево решений для сортировки вставками трех элементов. Чтобы понять его работу, рассмотрим сортировку вставками входного множества a = (3, 1, 2). Поскольку a1 ≥ a2, чтобы упорядочить эти два элемента, их нужно поменять местами. Затем алгоритм сравнивает последний элемент упорядоченного массива (исходное входное значение a1) со значением a3. Если a1 ≥ a3, то в последнем сравнении значения a3 с первым элементом отсортированной части (исходное входное значение a2) решается, куда поместить значение a2 в упорядоченном списке — на первое или второе место.

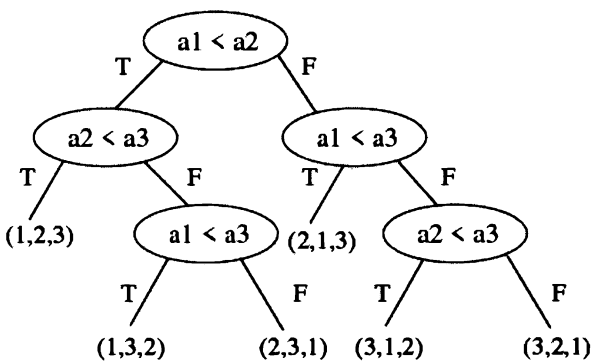



**Рис. 4.9.** Демонстрация работы упорядочивания элементов массива **a** сортировкой вставками с помощью дерева решений. Каждый узел дерева представляет конкретную входную перестановку, а путь от корня к концевому узлу описывает последовательность сравнений, выполняемых алгоритмом для ее упорядочивания

Эта нижняя граница важна по нескольким причинам. Прежде всего, такой подход можно расширить, чтобы получить нижние границы для многих приложений сортировки, включая определение уникальности элемента, поиск наиболее часто встречаю-

щегося элемента, а также создание выпуклых оболочек. Среди алгоритмических задач сортировка является одной из немногих, обладающих нетривиальной нижней границей. В *главе 11* мы рассмотрим другой подход к доказательству маловероятности существования быстрых алгоритмов.

Обратите внимание на то, что алгоритмы на основе хеширования не выполняют такого сравнения элементов, а помещают их вне пределов этой нижней границы. Но таким алгоритмам может не повезти, и в случае наихудшего случая невезения время исполнения любого рандомизированного алгоритма для одной из таких задач будет $\Omega(n \log n)$.

## 4.8. История из жизни. Скиена в суде

Я веду тихий, достаточно честный образ жизни. Одной из наград за такой образ жизни является то, что сплю я спокойно, не опасаясь никаких неприятных сюрпризов. Поэтому я был крайне поражен, когда мне позвонила женщина-адвокат, которая хотела не просто поговорить со мной, но поговорить об алгоритмах сортировки.

Ее фирма работала над делом, касающимся высокопроизводительных программ сортировки, и им был нужен эксперт, который мог бы объяснить присяжным технические подробности. Они узнали, что я кое-что понимаю в алгоритмах, и поэтому решили обратиться именно ко мне. Но прежде, чем нанять меня, они уточнили, как студенты оценивают мои преподавательские способности, чтобы удостовериться в том, что я могу доходчиво объяснять людям сложные понятия[6]. Участие в этом деле обещало стать увлекательной возможностью узнать, как *действительно* работают быстрые программы сортировки. Я полагал, что смогу ответить на вопрос, какой из алгоритмов сортировки в памяти является самым быстрым. Будет это пирамидальная сортировка или быстрая сортировка? Какие особенности алгоритмов способствовали сведению к минимуму количества сравнений в практических приложениях?

Ответ был довольно отрезвляющим: *сортировка в памяти никого не интересовала.* Главное заключалось в сортировке *громадных* файлов — намного больших, чем те, что могли поместиться в оперативную память. Основная работа состояла в записывании и считывании данных с диска, и хитроумные алгоритмы сортировки в памяти отнюдь не являлись здесь узким местом, поскольку в реальной жизни нужно было одновременно сортировать многие гигабайты данных.

Вспомним, что время обращения к жесткому диску относительно велико из-за необходимости позиционировать магнитную головку чтения/записи. Когда же головка установлена в нужном месте, передача данных осуществляется очень быстро, и для чтения большого блока данных требуется приблизительно такое же время, что и для одного байта. Таким образом, целью является сведение к минимуму количества блоков данных для чтения/записи и координация этих операций, чтобы алгоритм сортировки никогда не простаивал, ожидая данные.

---

[6] Один мой цинично настроенный коллега по профессорско-преподавательскому составу сказал, что это первый случай, когда кто-либо поинтересовался мнением студентов относительно преподавателей.

Необходимость интенсивной работы с диском во время сортировки лучше всего демонстрируется на ежегодных соревнованиях по сортировке *Minutesort*. Перед участниками ставится задача отсортировать наибольший объем данных за одну минуту. На момент подготовки этой книги чемпионом в этом соревновании является Tencent Sort, который упорядочил 55 терабайт данных чуть меньше чем за минуту на небольшом стареньком кластере из 512 узлов, каждый из которых оснащен 20 ядрами и 512 Гбайт оперативной памяти. Информацию о текущих рекордах сортировки можно получить на веб-сайте Sort Benchmark по адресу: **http://sortbenchmark.org/**.

Итак, какой алгоритм лучше всего подходит для сортировки данных вне оперативной памяти? Оказывается, это сортировка многоканальным слиянием с применением множества инженерных и других специальных приемов. Создается пирамида из членов верхнего блока каждого из $k$ отсортированных списков. Последовательно снимая с этой пирамиды верхний элемент и сливая вместе эти $k$ списков, алгоритм создает общий отсортированный список.

Поскольку пирамида находится в памяти, то эти операции выполняются с высокой скоростью. Когда набирается достаточно большой объем отсортированных данных, они записываются на диск, тем самим освобождая память для новых данных. Когда начинают заканчиваться элементы верхнего блока одного из сливаемых отсортированных $k$ списков, загружается следующий верхний блок данных из этого списка.

Оценить на этом уровне производительность программ/алгоритмов сортировки и решить, какой из них *действительно* быстрее, очень трудно. Будет ли справедливо сравнивать производительность коммерческой программы, предназначенной для обработки общих файлов, с производительностью кода, в котором убрано все лишнее и который оптимизирован для обработки целых чисел? В соревновании *Minutesort* в качестве входных данных для сортировки используются произвольно генерируемые записи размером в 100 байтов. Сортировка подобных записей существенно отличается от сортировки имен или целых чисел, поскольку имена — например, множественные экземпляры фамилии Shifflett, — не распределены случайным образом. Например, при сортировке этих записей широко применяется прием, при котором от каждого элемента берется короткий префикс, и сортировка сначала выполняется только по этим префиксам, просто чтобы избежать перемещения лишних байтов.

Какие же уроки я извлек из всего этого?

Самый важный состоит в том, что следует всячески избегать втягивания себя в судебное разбирательство в качестве как истца, так и ответчика.

Суды не являются инструментом для скорого разрешения разногласий. Юридические баталии во многом похожи на военные битвы: они весьма быстро обостряются, становятся очень дорогостоящими в денежном и временном отношении, а также в отношении душевного состояния, и обычно завершаются только тогда, когда обе стороны истощены и идут на компромисс. Мудрые люди могут решить свои проблемы, не прибегая к помощи судов. Усвоив этот урок должным образом сейчас, вы сможете сэкономить средства, в тысячи раз превышающие стоимость этой книги.

Что касается технических аспектов, то важно уделять должное внимание производительности внешних средств хранения данных при обработке очень больших наборов

данных алгоритмами с низкой временной сложностью (например, линейной или равной $\Theta(n \log n)$.

В подобных случаях даже такие постоянные множители, как 5 или 10, могут означать разницу между возможностью и невозможностью сортировки.

Конечно же, для наборов данных большого объема алгоритмы с квадратичным временем исполнения обречены на неудачу независимо от времени доступа к данным на дисках.

# Замечания к главе

В этой главе мы не рассмотрели несколько интересных алгоритмов сортировки, включая *алгоритм сортировки методом Шелла* (shellsort), представляющий собой более эффективную версию алгоритма сортировки вставками, и *алгоритм поразрядной сортировки* (radix sort), являющийся эффективным алгоритмом для сортировки строк. Узнать больше об этих и всех других алгоритмах сортировки можно в книге [Knu98].

В том виде, в каком он описан в этой главе, алгоритм сортировки слиянием копирует сливаемые элементы во вспомогательный буфер, чтобы не потерять оригинальные значения сортируемых элементов. Посредством сложных манипуляций с буфером этот алгоритм можно реализовать для сортировки элементов массива, не используя слишком большие объемы дополнительной памяти. В частности, алгоритм Кронрода (Kronrod) для слияния в памяти рассматривается в книге [Knu98].

Рандомизированные алгоритмы рассматриваются более подробно в книгах [MR95] и [MU17]. Задача подбора болтов и гаек была впервые представлена в книге [Raw92]. Сложный, но детерминированный алгоритм для ее решения с временной сложностью $\Theta(n \log n)$ рассматривается в книге [KMS98].

# 4.9. Упражнения

## Применение сортировки: сортировка чисел

1. *[3]* Вам нужно разделить $2n$ игроков на две команды по $n$ игроков в каждой. Каждому игроку присвоен числовой рейтинг, определяющий его игровые способности. Нужно разделить игроков наиболее *несправедливым* способом — т. е. создать самое большое неравенство игровых способностей между командой $A$ и командой $B$. Покажите, как можно решить эту задачу за время $O(n \log n)$.

2. *[3]* Для каждой из следующих задач предоставьте алгоритм, который находит требуемые числа за определенное время. Чтобы уменьшить объем решений, можете свободно использовать алгоритмы из этой книги в качестве процедур. Например, для множества $S = \{6, 13, 19, 3, 8\}$ разность $19 - 3$ максимальна, а разность $8 - 6$ — минимальна.

   а) Пусть $S$ — *неотсортированный* массив $n$ целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наибольшей разностью $|x - y|$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n)$.

b) Пусть $S$ — *отсортированный* массив $n$ целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наибольшей разностью $|x - y|$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(1)$.

c) Пусть $S$ — *неотсортированный* массив $n$ целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наименьшей разностью $|x - y|$ для $x \neq y$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n \log n)$.

d) Пусть $S$ — *отсортированный* массив $n$ целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наименьшей разностью $|x - y|$ для $x \neq y$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n)$.

3. *[3]* Для входного списка из $2n$ действительных чисел разработайте алгоритм с временем исполнения $O(n \log n)$, который разбивает эти числа на $n$ пар таким образом, чтобы минимизировать максимальную сумму значений в парах. Например, рассмотрим множество чисел {1, 3, 5, 9}. Эти числа можно разбить на следующие наборы: ({1, 3},{5, 9}), ({1, 5},{3, 9}) и ({1, 9},{3, 5}). Суммы значений пар в этих наборах равны (4, 14), (6, 12) и (10, 8). Таким образом, в третьем наборе максимальная сумма равна 10, что является минимумом для всех наборов.

4. *[3]* Допустим, что у нас есть $n$ пар элементов, где первый член пары является числом, а второй — одним из трех цветов: красный, синий или желтый. Эти пары элементов отсортированы по числу. Разработайте алгоритм с временем исполнения $O(n)$ для сортировки пар элементов по цвету (красный предшествует синему, а синий — желтому) таким образом, чтобы сохранить сортировку по числам для одинаковых цветов.

   Например: последовательность (1, синий), (3, красный), (4, синий), (6, желтый), (9, красный) после сортировки будет такой: (3, красный), (9, красный), (1, синий), (4, синий), (6, желтый).

5. *[3] Модой* набора чисел называют число с наибольшим количеством вхождений в набор. Например, модой набора {4, 6, 2, 4, 3, 1} является число 4. Разработайте эффективный алгоритм поиска моды набора из $n$ чисел.

6. *[3]* Дано: два набора элементов: $S_1$ и $S_2$ (оба размером $n$) и число $z$. Опишите алгоритм с временем исполнения $O(n \log n)$ для определения, существует ли пара элементов: один из набора $S_1$, а другой из набора $S_2$, сумма которых равна $x$. (Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $\Theta(n^2)$.)

7. *[5]* Разработайте эффективный алгоритм, принимающий в качестве ввода массив значений (неотрицательных целых чисел) количеств цитирования исследовательских работ и вычисляющий $h$-индекс для каждого исследователя. Согласно определению ученый имеет индекс $h$, если $h$ из его/её $n$ статей цитируются как минимум $h$ раз каждая, в то время как оставшиеся $(n - h)$ статей цитируются не более чем $h$ раз каждая.

8. *[3]* Предложите краткое описание метода для решения каждой из следующих задач. Укажите степень сложности в наихудшем случае для каждого из ваших методов.

   a) Вам дали огромное количество телефонных счетов и столь же огромное количество чеков по оплате этих счетов. Узнайте, кто не оплатил свой телефонный счет.

   b) Вам дали распечатанный список всех книг школьной библиотеки, в котором указаны название каждой книги, ее автор, идентификационный номер и издательство. Также вам дали список 30 издательств. Узнайте, сколько книг в библиотеке были выпущены каждым издательством.

c) Вам дали все карточки регистрации выдачи книг из библиотеки института за последний год, на каждой из которых указаны имена читателей, бравших книгу. Определите, сколько людей брали из библиотеки хотя бы одну книгу.

9. *[5]* Имеется набор $S$, содержащий $n$ целых чисел, и целое число $T$. Разработайте алгоритм с временем исполнения $O(n^{k-1}\log n)$ для определения, равна ли сумма $k$ целых чисел из $S$ целому числу $T$.

10. *[3]* Имеется набор $S$, содержащий $n$ действительных чисел, и действительное число $x$. Разработайте эффективный алгоритм для определения, содержит ли набор $S$ два таких элемента, сумма которых равна $x$.

    a) Допустим, что набор $S$ не отсортирован. Разработайте алгоритм для решения задачи за время $O(n \log n)$.

    b) Допустим, что набор $S$ отсортирован. Разработайте алгоритм для решения задачи за время $O(n)$.

11. *[8]* Разработайте алгоритм с временем исполнения $O(n)$, позволяющий найти все элементы, входящие больше чем $n/2$ раза в список из $n$ элементов. *Затем* разработайте алгоритм с временем исполнения $O(n)$, позволяющий найти все элементы, входящие в список из $n$ элементов больше чем $n/4$ раза.

## Применение сортировки: интервалы и множества

12. *[3]* Разработайте эффективный алгоритм для вычисления объединения множеств $A$ и $B$, где $n = \max(|A|,|B|)$. Выход должен быть представлен в виде массива отдельных элементов, образующих объединение множеств.

    a) Допустим, что $A$ и $B$ — неотсортированные массивы. Разработайте алгоритм для решения задачи за время $O(n \log n)$.

    b) Допустим, что $A$ и $B$ — отсортированные массивы. Разработайте алгоритм для решения задачи за время $O(n)$.

13. *[5]* Камера над дверью отслеживает время входа $a_i$ и время выхода $b_i$ (предполагается, что $b_i > a_i$) каждого из $n$ участников вечеринки. Разработайте алгоритм с временной сложностью $O(n \log n)$, который анализирует эти данные, чтобы определить период времени, когда на вечеринке одновременно присутствовало наибольшее количество участников. Можно предположить, что все значения времени входа и выхода разные.

14. *[5]* Имеется список $I$, содержащий $n$ интервалов в виде пар $(x_i, y_i)$. Разработайте алгоритм с временной сложностью $O(n \log n)$ в наихудшем случае, возвращающий список, в котором перекрывающиеся интервалы объединяются в один. Например, для входного множества $I = \{(1, 3), (2, 6), (8, 10), (7, 18)\}$ результат должен быть $\{(1, 6), (7, 18)\}$.

15. *[5]* Имеет набор $S$ из $n$ интервалов на линии, где $i$-интервал определяется его левой и правой крайними точками $(l_i, r_i)$. Разработайте алгоритм с временной сложностью $O(n \log n)$ для определения точки $p$ на линии, которая находится в наибольшем количестве интервалов. Например, набор из четырех интервалов $S = \{(10, 40), (20, 60), (50, 90), (15, 70)\}$ не содержит такой точки для всех четырех интервалов, но точка $p = 50$ является общей для трех интервалов. Можно предполагать, что конечная точка интервала находится в этом интервале.

16. *[5]* Имеется набор $S$ из $n$ отрезков линии, где отрезок $S_i$ определяется левой и правой крайними точками $(l_i, r_i)$. Разработайте эффективный алгоритм для выборки наимень-

шего количества сегментов, объединение которых полностью покрывает интервал от 0 до $m$.

## Пирамиды

17. *[3]* Разработайте алгоритм для поиска $k$ наименьших элементов в неотсортированном наборе из $n$ целых чисел за время $O(n + k \log n)$.

18. *[5]* Разработайте алгоритм с временем исполнения $O(n \log k)$ для слияния $k$ отсортированных списков с общим количеством $n$ элементов в один отсортированный список. (Подсказка: используйте пирамиду, чтобы ускорить работу простейшего алгоритма со временем исполнения $O(kn)$.)

19. *[5]* Вы можете сохранить набор из $n$ чисел в виде невозрастающей бинарной пирамиды или отсортированного массива. Для каждой из приведенных далее задач укажите, какая из этих структур данных является лучшей или не имеет значения, какую из них использовать. Обоснуйте свои ответы.

    a) Найти наибольший элемент.

    b) Удалить элемент.

    c) Сформировать структуру.

    d) Найти наименьший элемент.

20. [5]

    a) Разработайте эффективный алгоритм для поиска второго по величине элемента из $n$ элементов. Задача решается меньше чем за $2n - 3$ сравнений.

    b) Затем разработайте эффективный алгоритм для поиска третьего по величине элемента из $n$ элементов. Сколько операций сравнения выполняет ваш алгоритм в наихудшем случае? Приходится ли вашему алгоритму в процессе работы находить максимальный и второй по величине элементы?

## Быстрая сортировка

21. *[3]* Используя применяемый в быстрой сортировке принцип разбиения основной задачи на меньшие подзадачи, разработайте алгоритм для определения срединного элемента (median) массива $n$ целых чисел с ожидаемым временем исполнения $O(n)$. (Подсказка: нужно ли исследовать обе стороны раздела?)

22. *[3]* Срединным (median) элементом $n$ значений является $\lceil n/2 \rceil$-е наименьшее значение.

    a) Допустим, что алгоритм быстрой сортировки всегда выбирает в качестве элемента-разделителя срединное значение текущего подмассива. Сколько операций сравнений выполнит алгоритм быстрой сортировки в наихудшем случае при таком условии?

    b) Допустим, что алгоритм быстрой сортировки всегда выбирает в качестве элемента-разделителя $\lceil n/3 \rceil$-е наименьшее значение текущего подмассива. Сколько операций сравнений выполнит алгоритм быстрой сортировки в наихудшем случае при таком условии?

23. *[5]* Дан массив $A$ из $n$ элементов, каждый из которых окрашен в один из трех цветов: красный, белый или синий. Нужно отсортировать элементы по цвету в следующем порядке: красные, белые, синие. Разрешены только две операции:

    - `examine(A,i)` — возвращает цвет $i$-го элемента массива $A$;
    - `swap(A,i,j)` — меняет местами $i$-й и $j$-й элементы массива $A$.

    Создайте эффективный алгоритм сортировки элементов в указанном порядке за линейное время.

24. *[3]* Предоставьте эффективный алгоритм для упорядочивания $n$ элементов таким образом, чтобы все отрицательные элементы находились перед всеми положительными элементами. Использование вспомогательного массива для временного хранения элементов не разрешается. Определите время исполнения вашего алгоритма.

25. *[5]* Входной массив, который нужно упорядочить, содержит пару разных элементов — скажем: $z_i$ и $z_j$. Каким может быть наибольшее количество сравнений этих элементов друг с другом при использовании метода быстрой сортировки (quicksort)?

26. *[5]* Глубина рекурсии быстрой сортировки определяется как максимальное количество последовательных рекурсивных вызовов, выполняющихся до достижения базового элемента. Каковы минимальная и максимальная возможные глубины рекурсии для рандомизированного алгоритма быстрой сортировки?

27. *[8]* Перестановку $p$ целых чисел от 1 до $n$ нужно упорядочить в возрастающем порядке [1, ..., $n$]. Доступна только операция `reverse(p,i,j)`, которая меняет на обратный порядок элементов подпоследовательности $p_i$, ..., $p_j$ в перестановке. Например, чтобы упорядочить перестановку [1, 4, 3, 2, 5] достаточно одной смены порядка элементов на обратный (второго по четвертый элемент).

    - Докажите возможность упорядочивания любой перестановки, используя $O(n)$ реверсирований.
    - Теперь предположим, что затратность операции `reverse(p,i,j)` равна количеству элементов подпоследовательности: $|j - i| + 1$. Разработайте алгоритм для сортировки перестановки $p$ с затратностью, равной $O(n \log 2\ n)$. Проанализируйте время исполнения и затратность своего алгоритма и предоставьте доказательство его правильности.

## Сортировка слиянием

28. *[5]* Алгоритм сортировки слиянием модифицирован следующим образом: входной массив разбивается не на две, а на три части, рекурсивно сортируется каждая третья часть и результаты объединяются посредством процедуры слияния трех частей. Каким будет время исполнения в наихудшем случае модифицированного таким образом алгоритма сортировки слиянием?

29. *[5]* Дано $k$ упорядоченных массивов, каждый из которых содержит $n$ элементов. Требуется объединить их в один массив, содержащий $kn$ элементов. Один из подходов к решению этой задачи — многократно использовать процедуру слияния, объединив сначала первые два массива, затем объединив полученный результат с третьим массивом, и т. д., пока не будет объединен последний, $k$-й, массив. Какова временная сложность такого алгоритма?

30. *[5]* Снова рассмотрим задачу слияния $k$ массивов, каждый размером в $n$ элементов, в один упорядоченный массив размером в $kn$ элементов. Но на этот раз для слияния используем алгоритм, который сначала разбивает $k$ массивов на $k/2$ пар массивов, а затем посредством процедуры слияния объединяет каждую пару, создавая $k/2$ массивов размером $2n$ каждый. Этот шаг повторяется до тех пор, пока не останется один упорядоченный массив размером $kn$. Каким будет время исполнения такого алгоритма в зависимости от значений $n$ и $k$?

## Другие алгоритмы сортировки

31. *[5]* *Устойчивыми* называются такие алгоритмы сортировки, которые оставляют элементы с одинаковыми ключами в таком же порядке, в каком они находились до сортировки. Объясните, что нужно сделать, чтобы обеспечить устойчивость алгоритма сортировки слиянием.

32. *[5]* Сортировка массива волной (wiggle sort): требуется переупорядочить неотсортированный массив $A$ таким образом, чтобы $A[0] < A[1] > A[2] < A[3]$... Например, одно из возможных решений для ввода [3, 1, 4, 2, 6, 5] будет [1, 3, 2, 5, 4, 6]. Возможно ли выполнить такую сортировку за время $O(n)$, используя только $O(1)$ памяти?

33. *[3]* Продемонстрируйте, что $n$ положительных целых чисел в диапазоне от 1 до $k$ можно отсортировать за время $O(n \log k)$. Интересен случай $k \ll n$.

34. *[5]* Нам нужно отсортировать последовательность $S$ из $n$ целых чисел, содержащую много дубликатов. Количество различных целых чисел в $S$ равно $O(\log n)$. Разработайте алгоритм для сортировки таких последовательностей с временем исполнения в наихудшем случае $O(n \log \log n)$.

35. *[5]* Пусть $A[1...n]$ — массив, в котором первые $n - \sqrt{n}$ элементов уже отсортированы. О порядке остальных элементов нам ничего не известно. Разработайте алгоритм для сортировки массива $A$ за значительно лучшее время, чем $n \log n$.

36. *[5]* Допустим, что массив $A[1...n]$ может содержать числа из множества $\{1, ..., n^2\}$, но в действительности содержит самое большее $\log \log n$ этих чисел. Разработайте алгоритм для сортировки массива $A$ за время значительно меньшее, чем $O(n \log n)$.

37. *[5]* Необходимо отсортировать последовательность нулей (0) и единиц (1) посредством сравнений. При каждом сравнении значений $x$ и $y$ алгоритм определяет, какое из следующих отношений имеет место: $x < y$, $x = y$ или $x > y$.

    a) Разработайте алгоритм для сортировки этой последовательности за $n - 1$ сравнений в наихудшем случае. Докажите, что ваш алгоритм является оптимальным.

    b) Разработайте алгоритм для сортировки этой последовательности за $2n/3$ сравнений в среднем случае (допуская, что каждый из $n$ элементов может быть 0 или 1 с одинаковой вероятностью). Докажите, что ваш алгоритм является оптимальным.

38. *[6]* Пусть $P$ — простой, но не обязательно выпуклый, $n$-сторонний многоугольник, а $q$ — произвольная точка, не обязательно находящаяся в $P$. Разработайте эффективный алгоритм поиска прямой линии, начинающейся в точке $q$ и пересекающей наибольшее количество ребер многоугольника $P$. Иными словами, в каком направлении нужно целиться из ружья, находясь в точке $q$, чтобы пуля пробила наибольшее количество стен? Прохождение пули через одну из вершин многоугольника $P$ засчитывается как проби-

вание только одной стены. Для решения этой задачи возможно создание алгоритма с временем исполнения $O(n \log n)$.

## Нижние пределы

39. *[5]* В одной из моих научных статей (см. книгу [Ski88]) я привел пример алгоритма сортировки методом сравнений с временной сложностью $O(n \log(\sqrt{a}))$.

    Почему такой алгоритм оказался возможным, несмотря на то, что нижний предел сортировки равен $\Omega(n \log n)$?

40. *[5]* Один из ваших студентов утверждает, что он разработал новую структуру данных для очередей с приоритетами, которая поддерживает операции `insert`, `maximum` и `extract-max` с временем исполнения в наихудшем случае $O(1)$ для каждой. Докажите, что он ошибается. (Подсказка: для доказательства просто подумайте, каким образом такое время исполнения отразится на нижнем пределе времени исполнения сортировки, равном $\Omega(n \log n)$.)

## Поиск

41. *[3]* База данных содержит записи о 10 000 клиентов в отсортированном порядке. Из них сорок процентов считаются хорошими клиентами, т. е. на них приходится в сумме 60% обращений к базе данных. Такую базу данных и поиск в ней можно реализовать двумя способами:

    - поместить все записи в один массив и выполнять поиск требуемого клиента посредством двоичного поиска;
    - поместить хороших клиентов в один массив, а остальных — в другой. Двоичный поиск сначала выполняется в первом массиве, и только в случае отрицательного результата — во втором.

    Выясните, какой из этих подходов дает лучшую ожидаемую производительность. Будут ли результаты иными, если в обоих случаях вместо двоичного поиска применить линейный поиск в неотсортированном массиве?

42. *[5] Число Рамануджана — Харди* — это число, представимое в виде суммы двух кубов двумя различными способами. Иными словами, существуют четыре разных числа $a$, $b$, $c$ и $d$, для которых $a^3 + b^3 = c^3 + d^3$. Например, 1729 — число Рамануджана — Харди, поскольку $1729 = 13 + 123 = 93 + 103$.

    a) Разработайте эффективный алгоритм для проверки, является ли то или иное целое число числом Рамануджана — Харди, предоставив анализ сложности алгоритма.

    b) Разработайте эффективный алгоритм для создания всех чисел Рамануджана — Харди в диапазоне от 1 до $n$, предоставив анализ его сложности.

## Задачи по реализации

43. *[5]* Возьмем двумерный массив $A$ размером $n \times n$, содержащий целые числа (положительные, отрицательные и ноль). Допустим, что элементы в каждой строке этого массива отсортированы в строго возрастающем порядке, а элементы каждого столбца — в строго убывающем. (Соответственно, строка или столбец не может содержать двух

нулей.) Опишите эффективный алгоритм для подсчета вхождений элемента 0 в массив *A*. Выполните анализ времени исполнения этого алгоритма.

44. *[6]* Реализуйте несколько различных алгоритмов сортировки — таких как сортировка методом выбора, сортировка вставками, пирамидальная сортировка, сортировка слиянием и быстрая сортировка. Экспериментальным путем оцените сравнительную производительность этих алгоритмов в простом приложении, считывающим текстовый файл большого объема и отмечающим только один раз каждое встречающееся в нем слово. Это приложение можно реализовать, отсортировав все слова в тексте, после чего просканировав отсортированную последовательность для определения одного вхождения каждого отдельного слова. Напишите краткий доклад с вашими выводами.

45. *[5]* Реализуйте алгоритм внешней сортировки, использующий промежуточные файлы для временного хранения файлов, которые не помещаются в оперативную память. В качестве основы для такой программы хорошо подходит алгоритм сортировки слиянием. Протестируйте свою программу как на файлах с записями малого размера, так и на файлах с записями большого размера.

46. *[8]* Разработайте и реализуйте алгоритм параллельной сортировки, распределяющий данные по нескольким процессорам. Подходящим алгоритмом будет разновидность алгоритма сортировки слиянием. Оцените ускорение работы алгоритма с увеличением количества процессоров. Затем сравните время исполнения этого алгоритма с временем исполнения реализации чисто последовательного алгоритма сортировки слиянием. Каковы ваши впечатления?

## Задачи, предлагаемые на собеседовании

47. *[3]* Какой алгоритм вы бы использовали для сортировки миллиона целых чисел? Сколько времени и памяти потребует такая сортировка?

48. *[3]* Опишите преимущества и недостатки наиболее популярных алгоритмов сортировки.

49. *[3]* Реализуйте алгоритм, который возвращает только однозначные элементы массива.

50. *[5]* Как отсортировать файл размером в 500 Гбайт на компьютере, оснащенном оперативной памятью размером всего лишь в 4 Гбайт?

51. *[5]* Разработайте стек, поддерживающий выполнение операций занесения в стек, снятия со стека и извлечения наименьшего элемента. Каждая операция должна иметь постоянное время исполнения. Возможна ли реализация стека, удовлетворяющего этим требованиям?

52. *[5]* Дана строка из трех слов. Найдите наименьший (т. е. содержащий наименьшее количество слов) отрывок документа, в котором присутствуют все три слова. Предоставляются индексы расположения этих слов в строках поиска, например *word1: (1,4,5)*, *word2: (3,9,10)* и *word3: (2,6,15)*. Все списки отсортированы.

53. *[6]* Есть 12 монет, одна из которых тяжелее или легче, чем остальные. Найдите эту монету, выполнив лишь три взвешивания посредством балансирных весов.

## LeetCode

1. **https://leetcode.com/problems/sort-list/**
2. **https://leetcode.com/problems/queue-reconstruction-by-height/**

3. **https://leetcode.com/problems/merge-k-sorted-lists/**
4. **https://leetcode.com/problems/find-k-pairs-with-smallest-sums/**

## HackerRank

1. **https://www.hackerrank.com/challenges/quicksort3/**
2. **https://www.hackerrank.com/challenges/mark-and-toys/**
3. **https://www.hackerrank.com/challenges/organizing-containers-of-balls/**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org**:

1. «Vito's Family», глава 4, задача 10041.
2. «Stacks of Flapjacks», глава 4, задача 120.
3. «Bridge», глава 4, задача 10037.
4. «ShoeMaker's Problem», глава 4, задача 10026.
5. «ShellSort», глава 4, задача 10152.

ГЛАВА 5

# Метод «разделяй и властвуй»

Один из наиболее эффективных подходов к решению задач состоит в разбиении их на меньшие части, поддающиеся решению с большей легкостью. Задачи меньшего размера являются менее сложными, что позволяет фокусировать внимание на деталях, которые не попадают в поле зрения при исследовании всей задачи. Всякий раз, когда задачу можно разбить на более мелкие экземпляры задачи этого же типа, становится очевидным использование для ее решения рекурсивного алгоритма. В настоящее время практически все компьютеры оснащаются многоядерными процессорами, и для эффективной параллельной обработки задачу необходимо разложить по крайней мере на столько меньших задач, сколько ядер содержит процессор компьютера.

Принцип разбиения задачи на меньшие части лежит в основе двух важных парадигм разработки алгоритмов. В частности, в *главе 10* обсуждается динамическое программирование. Суть этого метода состоит в удалении из задачи некоторого элемента, решении получившейся меньшей задачи и корректного возвращения удаленного элемента в найденное решение меньшей задачи. А в методе «разделяй и властвуй» задача рекурсивно разбивается на, скажем, две половины, каждая половина решается по отдельности, после чего решения каждой половины объединяются в общее решение.

Эффективный алгоритм получается в том случае, когда слияние решений половин занимает меньше времени, чем их решение. Классическим примером алгоритма типа «разделяй и властвуй» является алгоритм сортировки слиянием, рассмотренный в *разд. 4.5*. Слияние двух отсортированных списков, содержащих по $n/2$ элементов и полученных за время $O(n \lg n)$, занимает только линейное время.

Принцип «разделяй и властвуй» применяется во многих важных алгоритмах, включая сортировку слиянием, быстрое преобразование Фурье и умножение матриц методом Страссена. Но я нахожу этот принцип трудным для практической разработки алгоритмов иных, чем двоичный поиск и его разновидности. Возможность анализа алгоритмов типа «разделяй и властвуй» зависит от нашего умения решать рекуррентные соотношения, определяющие сложность таких рекурсивных алгоритмов. Поэтому в этой главе мы и рассмотрим методы для решения рекуррентностей.

## 5.1. Двоичный поиск и связанные с ним алгоритмы

Классическим примером алгоритма типа «разделяй и властвуй» является двоичный поиск. Алгоритм двоичного поиска позволяет осуществлять быстрый поиск отсортированных ключей в массиве $S$. Чтобы найти ключ $q$, мы сравниваем значение $q$ со сред-

ним ключом массива $S[n/2]$. Если значение ключа $q$ меньше, чем значение ключа $S[n/2]$, значит, этот ключ должен находиться в левой половине массива $S$, в противном случае он должен находиться в его правой половине. Рекурсивно повторяя этот процесс на половине, содержащей элемент $q$, мы находим его за $\lceil \lg n \rceil$ сравнений, что является большим улучшением по сравнению с ожидаемыми $n/2$ сравнениями при последовательном поиске. Реализация алгоритма двоичного поиска на языке C приведена в листинге 5.1.

**Листинг 5.1. Реализация алгоритма двоичного поиска**

```
int binary_search(item_type s[], item_type key, int low, int high) {
    int middle; /* Индекс среднего элемента */

    if (low > high) {
        return (-1); /* Ключ не найден */
    }

    middle = (low + high) / 2;

    if (s[middle] == key) {
        return(middle);
    }

    if (s[middle] > key) {
        return(binary_search(s, key, low, middle - 1));
    } else {
        return(binary_search(s, key, middle + 1, high));
    }
}
```

Вероятно, все это вы уже знаете. Но важно понимать, насколько быстрым является алгоритм двоичного поиска. Существует популярная детская игра «Двадцать вопросов». Суть этой игры заключается в том, что один из игроков загадывает слово, а второй пытается угадать его, задавая вопросы типа «да/нет». Если после 20 вопросов слово не отгадано, то выигрывает первый игрок, в противном случае — второй. Но в действительности второй игрок всегда находится в выигрышном положении, поскольку для угадывания слова он может применить стратегию двоичного поиска. Для этого он берет словарь, открывает его посередине, выбирает слово (например, «ночь») и спрашивает первого игрока, находится ли загаданное им слово перед словом «ночь». Процесс рекурсивно повторяется для соответствующей половины, пока слово не будет отгадано.

Так как стандартные словари содержат от 50 000 до 200 000 слов, то можно быть уверенным, что 20 попыток будет более чем достаточно.

## 5.1.1. Частота вхождения элемента

Вариантами двоичного поиска являются несколько интересных алгоритмов. Допустим, мы хотим подсчитать, сколько раз тот или иной ключ $k$ (например, «Skiena») встречается в заданном отсортированном массиве. Так как при сортировке все копии $k$ соби-

раются в один непрерывный блок, то задача сводится к поиску этого блока и последующего измерения его размера.

Представленная ранее процедура двоичного поиска позволяет найти индекс элемента $x$ в соответствующем блоке за время $O(\lg n)$. Естественный способ определения границ блока — это последовательная проверка элементов слева от $x$ до тех пор, пока не будет найден элемент, отличающийся от ключа, и повторение процесса проверки для элементов справа от $x$. Разница между индексами этих границ блока, увеличенная на единицу, и будет количеством вхождений элемента $k$ в данный набор данных.

Этот алгоритм исполняется за время $O(\lg n + s)$, где $s$ — количество вхождений ключа. Но если весь массив состоит из одинаковых ключей, то это время может ухудшиться до $\Theta(n)$. Алгоритм двоичного поиска можно ускорить, модифицировав его для поиска *границ* блока, содержащего элемент $k$ вместо самого $k$. Допустим, мы удалим проверку на равенство:

```
if (s[middle] == key) return(middle);
```

из реализации двоичного поиска в листинге 5.1 и для каждого неуспешного поиска вместо –1 будем возвращать индекс `high`. Теперь *любой* поиск станет заканчиваться неудачей по причине отсутствия проверки на равенство. При сравнении ключа с одинаковым элементом массива процесс поиска будет переходить в правую половину массива, останавливаясь в конце концов на правой границе блока одинаковых элементов. Левая граница блока определяется изменением направления двоичного сравнения на обратное и повторением поиска. Так как поиск выполняется за время $O(\lg n)$, то подсчет количества вхождений элемента занимает логарифмическое время, независимо от размера блока.

Модифицировав нашу процедуру двоичного поиска, чтобы в случае неуспешного поиска она возвращала не –1, а `(low+high)/2`, мы получим позицию между двумя элементами массива, которую должен был бы занимать ключ $k$. Этот вариант наводит на мысль о другом способе решения нашей проблемы длительного времени исполнения. В частности, можно выполнять поиск позиций ключей $k - \varepsilon$ и $k + \varepsilon$, где $\varepsilon$ — константа достаточно малого размера, чтобы гарантировать неудачу обоих поисков при отсутствии промежуточных ключей. Опять же, исполнение двух двоичных поисков занимает время $O(\log n)$.

## 5.1.2. Односторонний двоичный поиск

Теперь допустим, что у нас есть массив, заполненный последовательностью нулей, за которыми следует неограниченная последовательность единиц, и нужно найти границу между этими двумя последовательностями. Если бы мы знали количество $n$ элементов массива, то на определение точки перехода посредством двоичного поиска ушло бы $\lceil \lg n \rceil$ операций сравнения. Но при отсутствии такой границы мы можем последовательно выполнять сравнения по увеличивающимся интервалам ($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, ...), пока не найдем ненулевой элемент. Теперь у нас имеется окно, содержащее целевой элемент, и мы можем применить двоичный поиск. Такой *односторонний двоичный поиск* возвращает границу $p$ за самое большее $2\lceil \lg p \rceil$ операций сравнения, независимо от размера массива. Односторонний двоичный поиск лучше всего подхо-

дит для локализации элемента, расположенного недалеко от текущей позиции просмотра.

### 5.1.3. Корни числа

Квадратным корнем числа $n$ является такое положительное число $r$, для которого $r^2 = n$. Хотя операция вычисления квадратного корня имеется в любом карманном калькуляторе, нам будет полезно разработать эффективный алгоритм для его вычисления.

Заметим, что квадратный корень числа $n > 1$ должен находиться в интервале от 1 до $n$. Пусть $l = 1$, $r = n$. Теперь рассмотрим среднюю точку этого интервала $m = (l + r)/2$ и отношение $m^2$ к $n$. Если $n > m^2$, то квадратный корень должен быть больше, чем $m$, поэтому мы устанавливаем $l = m$ и повторяем процедуру. Если $n < m^2$, то квадратный корень должен быть меньшим, чем $m$, поэтому устанавливаем $r = m$ и повторяем процедуру. В любом случае мы сократили интервал наполовину посредством всего лишь одного сравнения. Продолжая действовать подобным образом, мы найдем квадратный корень с точностью до $\pm 1/2$ без учета знака за $\lceil \lg n \rceil$ сравнений.

Этот метод *деления интервала пополам* можно также применять для решения более общей задачи поиска корней уравнения. Число $x$ называется *корнем* функции $f$, если $f(x) = 0$. Возьмем два числа $l$ и $r$, для которых $f(l) > 0$ и $f(r) < 0$. Если $f$ является непрерывной функцией, то ее корень должен находиться в интервале между $l$ и $r$. В зависимости от знака $f(m)$, принимая $m = (l + r)/2$, мы можем уменьшить это содержащее корень окно наполовину за одно сравнение, прекращая поиск, как только наша оценка корня становится достаточно точной.

Для обоих типов задач поиска корня известны алгоритмы их решения, которые выдают результат быстрее, чем двоичный поиск. В частности, вместо исследования средней точки интервала в этих алгоритмах применяется интерполяция для поиска подходящей точки, расположенной ближе к искомому корню. Тем не менее метод двоичного поиска является простым и надежным и работает так хорошо, насколько это возможно, не требуя дополнительной информации о самой функции.

#### Подведение итогов

Двоичный поиск и его вариации являются классическими примерами алгоритмов типа «разделяй и властвуй».

## 5.2. История из жизни. Поиск «бага в баге»

Ютонг (Yutong) поднялся, чтобы огласить результаты многонедельной тяжелой работы. «Мертвый», — объявил он демонстративно.

Комнату заполнил коллективный стон. Я был членом команды, разрабатывающей новый способ создания вакцин — SAVE (Synthetic Attenuated Virus Engineering — проектирование синтетических ослабленных вирусов). Вследствие особенностей работы генетического кода кодирование любого белка длиной $n$ обычно осуществляется посредством приблизительно $3n$ возможных генов. С первого взгляда все эти гены кажутся одинаковыми, поскольку описывают один и тот же белок. Но каждый из этих

3$n$ одинаковых генов использует свои биологические средства несколько по-иному, выполняя преобразования с немного другой скоростью.

Мы надеялись создать вакцину, заменив какой-либо ген вируса менее опасной его версией, чтобы создать ослабленную версию вируса, которая во всех остальных аспектах была бы такой же, как и исходная. Это дало бы возможность иммунной системе натренироваться, так сказать, «на кошках»[1], на борьбу с более сильными версиями вируса без необходимости подвергать организм полномасштабной болезни. Но нам был нужен ослабленный вирус, а не мертвый, т. к. невозможно научиться каким-либо приемам борьбы, сражаясь с уже мертвым противником.

— «Мертвый» означает, что должно быть одно место в этой области из 1200 оснований, где в вирусе содержится значение последовательности, необходимой ему для выживания, — сказал наш главный вирусолог. — Изменив последовательность на этом этапе, мы убили вирус. И мы должны найти это место, чтобы вернуть его к жизни.

— Но оно может находиться в любой из 1200 точек последовательности! Как мы можем найти его? — спросил Ютонг.

Я поразмыслил немного над этим. Нам нужно было найти баг в баге[2]. Это похоже на задачу отладки программы. Я вспомнил множество тоскливых ночей, проведенных в попытках вычислить, какая именно строка программы вызывала в ней сбой. В таких случаях я часто прибегал к удалению из программы фрагментов кода, закомментировав их, чтобы проверить, будет ли программа работать нормально с оставшимся кодом. Обычно причину проблемы можно было легко обнаружить, постепенно уменьшая удаляемый таким образом фрагмент кода. Наилучшим подходом к поиску проблемного участка в вирусе будет...

— Двоичный поиск, — провозгласил я. Предположим, что мы заменим первую половину кодирующей последовательности жизнеспособного гена кодирующей последовательностью мертвого штамма гена с отсутствующим критическим местом (эти последовательности соответственно помечены зеленым и красным цветом в проекте II на рис. ЦВ-5.1).

Если такой гибридный ген окажется жизнеспособным, это будет означать, что критическое место должно находиться в правой половине гена, а мертвый ген будет означать, что проблема должна находиться в его левой половине. С помощью такого процесса двоичного поиска нужное место можно локализировать в одной из $n$ областей за $\lceil \log_2 n \rceil$ последовательных этапов эксперимента.

— Мы можем сузить содержащую критическое место область в гене длиной $n$ оснований до $n/16$ оснований, выполнив всего лишь четыре эксперимента, — сказал я. Главный вирусолог был вне себя от радости. А Ютонг побледнел.

— Еще четыре этапа экспериментов, — заныл он. — У меня ушел целый месяц, чтобы синтезировать, клонировать и вырастить этот вирус в последний раз. Теперь вы хотите,

---

[1] Российские читатели, несомненно, вспомнят Балбеса из комедии Леонида Гайдая «Операция "Ы" и другие приключения Шурика». — *Прим. ред.*

[2] Игра слов в исходном тексте: «Finding the bug in the bug» — слово «bug» может означать как ошибку в программе, так и инфекционный возбудитель типа вируса или бактерии. — *Прим. пер.*

чтобы я снова это сделал, потом дождался, чтобы узнать, в какой половине находится критическое место, а затем повторил весь процесс еще три раза? Забудьте об этом!

Ютонг понимал, что мощь двоичного поиска основывается на *взаимодействии*: задаваемый на этапе $r$ запрос зависит от ответов, полученных на запросы на этапах с 1 по $r - 1$. Двоичный поиск по своей природе является последовательным алгоритмом. И когда каждое отдельное сравнение выливается в длительный и трудоемкий процесс, количество сравнений величиной $\lg n$ перестает выглядеть таким привлекательным. Но у меня в рукаве был припрятан козырный туз.

— Четыре последовательных этапа — это слишком много работы для тебя, Ютонг. Но, возможно, ты сможешь выполнить четыре разных проекта одновременно, если мы дадим их тебе все вместе? — спросил я.

— Если работать над всеми четырьмя разными последовательностями одновременно, то это не столь большая проблема, — ответил он. — Не намного труднее, чем делать только одну из них.

Решив этот вопрос, я предложил, чтобы они одновременно синтезировали четыре проекта вируса, обозначенных I, II, III и IV на рис. ЦВ-5.1. Оказывается, двоичный поиск *можно* выполнять в параллельном режиме — при условии, что запросы могут быть к произвольным подмножествам, а не к связанным половинам. Обратите внимание на то, что каждый из столбцов, определяемый этими четырьмя проектами, состоит из отдельного набора красных и синих областей последовательности. Таким образом, местонахождение критической точки однозначно определяется набором «живой/мертвый» областей в этих четырех проектах синтетического вируса. В приведенном примере мертвым оказался вирус проекта I, тогда как остальные три были живыми, что позволило точно установить пятую справа область как местонахождение летальной точки.

Ютонг славно потрудился и после месяца (но не месяцев) упорного труда открыл критическое место в вирусе полиомиелита [SLW+12]. Он нашел «баг в баге» посредством метода «разделяй и властвуй», который лучше всего работает при разделении задачи на каждом этапе пополам. Обратите внимание на то, что все четыре наших проекта вируса содержат половину красных и половину зеленых областей, но упорядоченных шестнадцатью разными перестановками. При интерактивном двоичном поиске в конце сравниваются только две последние перестановки. Расширив каждое сравнение до работы с половиной последовательности, мы устранили необходимость в последовательных сравнениях, намного таким образом ускорив процесс.

## 5.3. Рекуррентные соотношения

Временна́я сложность многих алгоритмов типа «разделяй и властвуй» естественно формируется рекуррентными соотношениями. Умение решать рекуррентные соотношения является важной предпосылкой для понимания, в каких обстоятельствах от алгоритмов типа «разделяй и властвуй» можно ожидать хорошей производительности, а также представляет собой важный инструмент для общего анализа. Читатели, которых идея анализа не приводит в особый восторг, могут пропустить этот раздел, но должное

представление о разработке алгоритмов можно получить, лишь понимая поведение рекуррентных соотношений.

Что же собой представляет рекуррентное соотношение? Это уравнение, в котором функция определяется посредством самой себя. В качестве примера рекуррентного соотношения можно привести последовательность чисел Фибоначчи, определяемую равенством

$$F_n = F_{n-1} + F_{n-2}$$

вместе с начальными значениями $F_0 = 0$ и $F_1 = 1$ (см. *разд. 10.1.1*). С помощью рекуррентных соотношений можно выразить также многие другие знакомые нам функции. В частности, посредством рекуррентного соотношения можно представить любой многочлен — например, линейную функцию:

$$a_n = a_{n-1} + 1, a_1 = 1 \rightarrow a_n = n.$$

Любую показательную функцию также можно выразить посредством рекуррентного соотношения — например:

$$a_n = 2a_{n-1}, a_1 = 1 \rightarrow a_n = 2^{n-1}.$$

Наконец, многие необычные функции, которые непросто выразить посредством обычной нотации, можно представить натуральным образом с помощью рекуррентного соотношения — например:

$$a_n = na_{n-1}, a_1 = 1 \rightarrow a_n = n!.$$

Все это демонстрирует, что рекуррентные соотношения являются очень гибким средством для представления функций. Кроме рекуррентных соотношений свойством ссылаться на самих себя также обладают рекурсивные программы или алгоритмы, как можно видеть по общему корню обоих терминов. По существу, рекуррентные соотношения предоставляют способ анализировать рекурсивные структуры, такие как алгоритмы.

## 5.3.1. Рекуррентные соотношения метода «разделяй и властвуй»

Как уже упоминалось, типичный алгоритм типа «разделяй и властвуй» разбивает задачу на $a$ меньших подзадач, каждая из которых имеет размер $n/b$. Затем решения подзадач сливаются в общее решение, занимая время $f(n)$. Пусть $T(n)$ — время решения этим алгоритмом наихудшего случая задачи размером $n$. Тогда $T(n)$ предоставляется следующим рекуррентным соотношением:

$$T(n) = aT(n/b) + f(n).$$

Рассмотрим примеры использования рекуррентных соотношений, основанных на ранее рассмотренных алгоритмах, для решения следующих задач.

- *Сортировка слиянием.*

  Время исполнения алгоритма сортировки слиянием определяется рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, поскольку алгоритм рекурсивно разделяет входные данные на равные половины, после чего выполняет слияние частных решений

в общее за линейное время. Фактически это рекуррентное соотношение сводится к соотношению $T(n) = O(n \lg n)$, которое было получено ранее.

- *Двоичный поиск.*

  Время исполнения алгоритма двоичного поиска представляется рекуррентным соотношением $T(n) = T(n/2) + O(1)$, поскольку каждый шаг уменьшения размера задачи вдвое выполняется за линейное время. Фактически это рекуррентное соотношение сводится к соотношению $T(n) = O(\lg n)$, которое было получено ранее.

- *Быстрое создание пирамиды.*

  Процедура `bubble_down` (см. листинг 4.5) формирует пирамиду из $n$ элементов, создавая две пирамиды, каждая из которых содержит $n/2$ элементов, а потом сливая их с корнем. Эта процедура занимает логарифмическое время. Таким образом, время исполнения определяется рекуррентным соотношением $T(n) = 2T(n/2) + O(\lg n)$. Это соотношение сводится к соотношению $T(n) = O(n)$, которое было получено ранее.

Решение рекуррентного соотношения состоит в нахождении красивой закрытой формы, описывающей или ограничивающей результат. Для решения рекуррентных соотношений, обычно возникающих из алгоритмов типа «разделяй и властвуй», можно использовать *основную теорему*, рассматриваемую в *разд. 5.4*.

## 5.4. Решение рекуррентных соотношений типа «разделяй и властвуй» (*)

Рекуррентные соотношения типа «разделяй и властвуй» в форме

$$T(n) = aT(n/b) + f(n)$$

обычно очень легко решаются, поскольку их решения обычно относятся к одному из трех отдельных классов:

1. Если для некоторой константы $\varepsilon > 0$ существует функция $f(n) = O(n^{\log_b a - \varepsilon})$, тогда $T(n) = \Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, тогда $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Если для некоторой константы $\varepsilon > 0$ существует функция $f(n) = \Omega(n^{\log_b a + \varepsilon})$ и для некоторой константы $c < 1$ существует функция такая, что $af(n/b) \le cf(n)$, тогда $T(n) = \Theta(f(n))$.

Хотя все эти формулы выглядят устрашающе, в действительности их совсем не трудно использовать. Вопрос заключается в определении, какой случай так называемой *основной теоремы* является действительным для того или иного рекуррентного соотношения. Первый случай применим для создания пирамиды и умножения матриц, а второй — действителен для сортировки слиянием. Третий случай обычно нужен для не столь элегантных алгоритмов, в которых затраты на слияние подзадач превышают затраты на все остальные операции.

Основную теорему можно представлять себе в виде «черного ящика», которым мы умеем пользоваться, но устройство которого нам неизвестно. Однако после некоторого размышления понимание, как работает основная теорема, появляется.

На рис. 5.2 показано рекурсивное дерево для типичного алгоритма типа «разделяй и властвуй», выражаемого рекуррентным соотношением $T(n) = aT(n/b) + f(n)$.

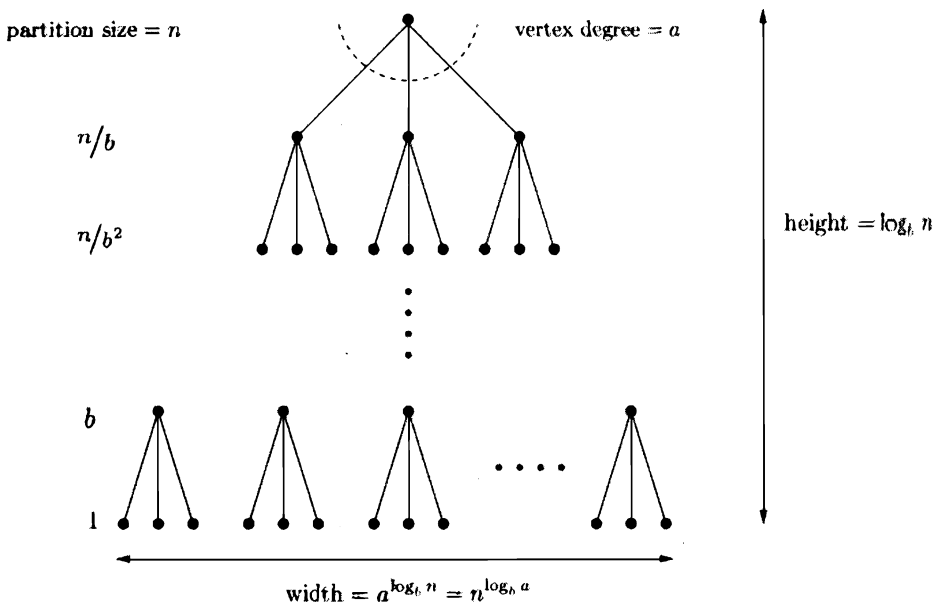



**Рис. 5.2.** Рекурсивное дерево, полученное в результате разложения каждой задачи размером $n$ на $a$ задач размером $n/b$

Задача размером в $n$ элементов разбивается на $a$ подзадач размером $n/b$ элементов. Каждая подзадача размером в $k$ элементов выполняется за время $O(f(k))$. Общее время исполнения алгоритма будет равно сумме временных затрат на выполнение этих подзадач, сложенной с накладными расходами на создание рекурсивного дерева. Высота этого дерева равна $h = \log_b n$, а количество листьев равно $a^h = a^{\log_b n}$. Посредством определенных алгебраических манипуляций последнее выражение можно упростить до $n^{\log_b a}$.

Три случая основной теоремы соответствуют трем разным типам временных затрат, каждый из которых может доминировать в зависимости от $a$, $b$ и $f(n)$:

1. *Слишком много листьев.*

   Если количество листьев превышает общие затраты на внутреннюю обработку, то общее время исполнения будет равно $O(n^{\log_b a})$.

2. *Одинаковый объем работы на каждом уровне.*

   По мере прохождения вниз по дереву размер каждой задачи уменьшается, но количество задач, подлежащих решению, увеличивается. Если сумма затрат на внутрен-

нюю обработку одинакова для каждого уровня, то общее время исполнения вычисляется умножением затрат на каждом уровне ( $n^{\log_b n}$ ) на количество уровней ($\log_b n$) и будет равно $O(n^{\log_b a} \lg n)$.

3. *Слишком большое время обработки корня.*

   Если с возрастанием $n$ затраты на внутреннюю обработку возрастают очень быстрыми темпами, то затраты на обработку корня будут доминировать над всем прочим. В таком случае общее время исполнения будет равно $O(f(n))$.

Убедившись в работоспособности основной теоремы, вы сможете с легкостью анализировать любой алгоритм типа «разделяй и властвуй», имея в своем распоряжении только связанное с ним рекуррентное соотношение. Далее мы применяем этот подход для нескольких алгоритмов.

## 5.5. Быстрое умножение

Мы знаем как минимум два способа, как умножить целые числа $A$ и $B$ друг на друга, чтобы получить их произведение $A \times B$. Сначала нам показали, что $A \times B$ означает сложение вместе $B$ копий $A$, что является алгоритмом с временной сложностью $O(n \cdot 10^n)$ для умножения двух $n$-цифровых десятичных чисел. Затем нас научили умножать большие числа поразрядно — например:

$$9256 \times 5367 = 9256 \times 7 + 9256 \times 60 + 9256 \times 300 + 9256 \times 5000 = 13\,787\,823.$$

Вспомним, что добавляемые к цифрам-множителям нули в действительности *не используются* в вычислении произведения. Их воздействие реализуется сдвигом цифры произведения влево на соответствующее количество позиций. Предполагая, что каждое поразрядное умножение выполняется за постоянное время простым выбором соответствующего значения в таблице умножения, этот алгоритм умножает два $n$-разрядных числа за время $O(n^2)$.

В этом разделе мы рассмотрим еще более быстрый алгоритм для умножения больших чисел, представляющий собой классический алгоритм типа «разделяй и властвуй». Предположим, что каждое из умножаемых чисел состоит из $n = 2m$ разрядов. Обратите внимание на то, что каждое число можно разделить на две части по $m$ разрядов каждая таким образом, что произведение полных чисел можно легко сконструировать из произведений этих частей, как показано далее. Пусть $w = 10^{m+1}$ и представляет $A = a_0 + a_1 w$ и $B = b_0 + b_1 w$, где $a_i$ и $b_i$ являются частями каждого соответствующего числа. Тогда

$$A \times B = (a_0 + a_1 w) \times (b_0 + b_1 w) = a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2.$$

Эта процедура сводит задачу умножения двух $n$-разрядных чисел к получению четырех произведений $(n/2)$-разрядных чисел. Вспомним, что умножение на $w$ в действительности не выполняется, а просто добавляет к произведению соответствующее количество нулей. Полученные четыре произведения нужно затем сложить вместе, на что требуется $O(n)$ времени.

Пусть $T(n)$ обозначает количество времени, занимаемое умножением двух $n$-разрядных чисел. В предположении, что рекурсивное применение того же самого алгоритма про-

изводится с каждым из меньших произведений, время исполнения этого алгоритма определяется следующим рекуррентным соотношением:

$$T(n) = 4T(n/2) + O(n).$$

Ориентируясь на *случай 1* основной теоремы, можно видеть, что время исполнения этого алгоритма равно $O(n^2)$ — точно такое же, как в случае поразрядного умножения. Разделить-то мы разделили, а вот властвовать не получилось.

Альтернативное рекуррентное соотношение для умножения, называемое *алгоритмом Карацубы*, дает лучшее время исполнения. Предположим, что нам нужно умножить следующие три числа:

$$q_0 = a_0 b_0;$$
$$q_1 = (a_0 + a_1)(b_0 + b_1);$$
$$q_2 = a_1 b_1.$$

Обратите внимание, что

$$A \times B = (a_0 + a_1 w) \times (b_0 + b_1 w) = a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2 = \\ = q_0 + (q_1 - q_0 - q_2)w + q_2 w^2.$$

То есть теперь мы вычислили полное произведение, используя только три умножения половин и постоянное количество сложений. Опять же, члены $w$ не участвуют в умножении, а просто обозначают сдвиг влево, или добавление соответствующего количества нулей. Поэтому временная сложность этого алгоритма определяется следующим рекуррентным соотношением:

$$T(n) = 3T(n/2) + O(n).$$

Поскольку

$$n = O\left(n^{\log_2 3}\right),$$

то это соотношение решается посредством первого случая основной теоремы и

$$T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1,588}\right).$$

Мы получили здесь серьезное улучшение по сравнению с алгоритмом с квадратичной сложностью для больших чисел, которое действительно значительно превосходит стандартный алгоритм умножения для чисел в 500 и больше разрядов.

Этот подход с определением рекуррентного соотношения, использующего меньшее количество умножений, но большее количество сложений, также прослеживается в основе быстрых алгоритмов для умножения матриц. Как показано в *разд. 2.5.4*, время исполнения алгоритма с вложенными циклами для умножения двух матриц размером $n \times n$ равно $O(n^3)$, т. к. для каждого из $n^2$ элементов в матрице произведений мы вычисляем скалярное произведение $n$ терминов. Но в книге [Str69] рассматривается алгоритм типа «разделяй и властвуй», который для умножения двух матриц размером $n \times n$ манипулирует произведениями семи матриц размером $n/2 \times n/2$.

Временная сложность этого рекуррентного соотношения равна

$$T(n) = 7 \cdot (n/2) + O(n^2).$$

Поскольку $\log_2 7 \approx 2.81$, и $O(n^{\log_2 7})$ доминирует над $O(n^2)$, то применим первый случай основной теоремы и получим: $T(n) = \Theta(n^{2.81})$.

Этот алгоритм подвергся постоянным «улучшениям» все более сложными рекуррентными соотношениями, и его текущее наилучшее время исполнения составляет $O(n^{2,3727})$. Дополнительные подробности приводятся в *разд. 16.3*.

## 5.6. Поиск наибольшего поддиапазона и ближайшей пары

Предположим, вам дали задачу создать рекламу для хедж-фонда, помесячная производительность которого в этом году была такой:

$$[-17, 5, 3, -10, 6, 1, 4, -3, 8, 1, -13, 4].$$

В целом за год фонд оказался убыточным, но доходы с мая по октябрь были наибольшими за какой-либо период, составляя в чистом итоге 17 единиц. Этот факт можно использовать в качестве положительного аспекта для рекламирования фонда.

Каким же образом определялся этот период? Задача нахождения поддиапазона с наибольшей суммой значений для входного массива $A$ из $n$ значений заключается в нахождении пары таких значений индекса $i$ и $j$, которые максимизируют значение $S = \sum_{k=i}^{j} A[k]$. Сложение всех значений массива не обязательно максимизирует $S$ вследствие наличия в нем отрицательных значений. А для явной проверки каждой возможной пары значений начала и конца интервала требуется время $\Omega(n^2)$. Далее приводится алгоритм типа «разделяй и властвуй» с временем исполнения, равным $O(n \log n)$.

Предположим, мы разделили массив $A$ на правую и левую половины. Поддиапазон с наибольшей суммой значений может находиться или в левой половине, или в правой, или же перекрывать смежные края этих половин. Рекурсивная программа для нахождения наибольшего диапазона между значениями $A[l]$ и $A[r]$ может легко вызывать саму себя для обработки левой и правой подзадач. Но как найти наибольший поддиапазон, перекрывающий смежные края левой и правой частей, т. е. проходящий через элементы $m$ (от *m*iddle — середина) и $m + 1$?

Ключ к решению этой задачи — наибольший поддиапазон по центру полного диапазона — будет объединением наибольшего поддиапазона в левой части с конечным элементом $m$ и наибольшего поддиапазона в правой части с начальным элементом $m + 1$, как показано на рис. ЦВ-5.3.

Значение $V_l$ такого наибольшего поддиапазона в левой части можно вычислить за линейное время следующей программой сканирования:

```
LeftMidMaxRange(A, l,m)
      S = M = 0
      for i = m downto l
            S = S + A[i]
            if (S > M) then M = S
      return S
```

Наибольший поддиапазон в правой части вычисляется аналогичным образом. Разбиение $n$ на две половины, выполнение линейных и рекуррентных вычислений занимает время:

$$T(n), \text{ где } T(n) = 2 \cdot T(n/2) + \Theta(n).$$

Второй случай основной теоремы дает время $T(n) = \Theta(n \log n)$.

Этот общий подход, состоящий в «находим наилучшее решение для каждой части, а затем проверяем, что у нас перекрывает середину», можно также применять и для решения других задач. Например, рассмотрим задачу нахождения наименьшего расстояния между парами точек в множестве из $n$ точек.

С одной стороны, это простая задача — как мы видели в *разд. 4.1*, после упорядочения точек ближайшей парой точек должны быть смежные точки. Таким образом, выполняя после сортировки сканирование слева направо, занимающее линейное время, получаем алгоритм с временной сложностью $\Theta(n \log n)$. Но сканирование можно заменить хитроумным алгоритмом типа «разделяй и властвуй». Ближайшая пара значений определяется левой или правой половиной точек или же парой точек посередине диапазона. Найти эти точки можно с помощью следующего алгоритма:

```
ClosestPair(A, l, r)
    mid = ⌊l + r/2⌋
    l_min = ClosestPair(A, l, mid)
    r_min = ClosestPair(A, mid + 1, r)
    return min(l_min, r_min, A[m + 1] - A[m])
```

Так как при каждом вызове этот алгоритм исполняет постоянную работу, его общее время исполнения определяется следующим рекуррентным соотношением:

$$T(n) = 2 \cdot T(n/2) + O(1).$$

Согласно второму случаю основной теоремы время $T(n) = \Theta(n)$.

Поскольку время исполнения остается линейным, это может выглядеть чем-то не слишком особенным, но давайте обобщим эту идею на точки в двух измерениях. После сортировки $n$ точек с координатами $(x, y)$ по значениям их $x$-координат они должны обладать таким же свойством, т. е. ближайшей парой точек будут или две точки в левой или правой половине, или же пара точек слева и справа от середины. Как можно видеть на рис. 5.4, в последнем случае эти две точки должны быть расположены близ-

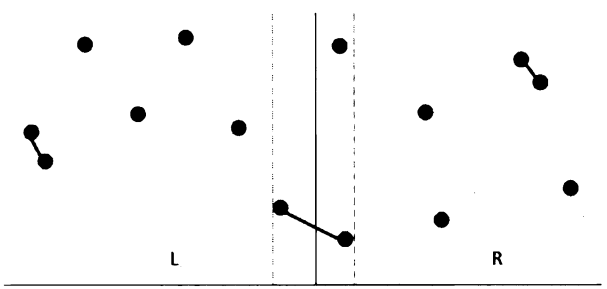



**Рис. 5.4.** Ближайшая пара точек в двумерном пространстве находится или слева, или справа от центра, или же в узкой центральной полосе

ко к разделяющей линии (расстояние $d < min(l_{min}, r_{min})$), а также иметь подобные $y$-координаты.

Применив ряд сложных вычислений, ближайшую пару точек, расположенных по обеим сторонам разделительной линии, можно найти за линейное время, что дает нам время исполнения, равное

$$T(n) = 2 \cdot T(n/2) + \Theta(n) = \Theta(n \log n),$$

как определяется во втором случае основной теоремы.

# 5.7. Параллельные алгоритмы

Ум хорошо, а два лучше, и, обобщая эту максиму, можно сделать вывод, что $n$ умов лучше, чем $n - 1$ умов. Это также относится и к вычислениям. Подтверждает наш вывод все более широкое применение параллельной обработки данных, ставшее возможным благодаря распространению многоядерных процессоров и кластерных вычислений.

## 5.7.1. Параллелизм на уровне данных

Парадигма разработки алгоритмов «разделяй и властвуй» лучше всего подходит для параллельных вычислений. Обычно мы пытаемся разделить решаемую задачу размером $n$ на $p$ частей одинакового размера, каждая из которых подается на вход отдельного процессора одновременно с другими частями. Таким образом время обработки (makespan) уменьшается с $T(n)$ до $T(n/p)$ плюс затраты на объединение частичных результатов в один. В случае линейного времени $T(n)$ этот подход дает нам максимально возможное ускорение вычислений $p$. Если $T(n) = \Theta(n_2)$, то может казаться, что вычисления можно ускорить еще больше, но это обычно только иллюзия. Предположим, что нам нужно просканировать все пары $n$ элементов. Конечно же, можно разделить все множество элементов на $p$ независимых частей, но $n^2 - p(n/p)^2$ из $n^2$ возможных пар никогда не будут иметь оба элемента на одном и том же процессоре.

Многоядерные процессоры обычно лучше всего применять для того, чтобы воспользоваться *параллелизмом на уровне данных*, исполняя один алгоритм на разных и независимых множествах данных. Например, чтобы получить реалистическую анимацию, компьютерные анимационные системы должны прорисовывать тридцать кадров в секунду. Возможно, наилучшим способом справиться с такой задачей вовремя может быть выделение отдельного процессора для обработки каждого кадра или для обработки одной части кадра. Подобные задачи часто называются *чрезвычайно параллельными* (embarrassingly parallel).

По большому счету такие подходы параллельной обработки данных не представляют интереса с алгоритмической точки зрения, но они простые и эффективные. Существует более продвинутая область параллельных алгоритмов, где усилия разных процессоров синхронизируются для совместного решения задачи за более короткое время, чем это мог бы сделать один процессор. Такие алгоритмы выходят за рамки рассматриваемой в этой книге тематики, но следует быть осведомленным о сложностях, связанных с разработкой и реализацией параллельных алгоритмов высокого уровня сложности.

## 5.7.2. Подводные камни параллелизма

Итак, параллельные алгоритмы имеют несколько скрытых недостатков, о которых нужно знать.

- *Потенциальный выигрыш часто невелик.*

  Допустим, у вас имеется исключительный доступ к компьютеру с двадцатью четырьмя процессорами. Теоретически, используя все эти процессоры, можно в 24 раза повысить скорость работы самой быстрой последовательной программы. Это, конечно же, прекрасно, но, возможно, производительность можно повысить, использовав более удачный последовательный алгоритм. Время, требуемое на переработку кода для исполнения на параллельных процессорах, возможно, лучше потратить на улучшение последовательной версии программы. Также нужно иметь в виду, что средства отладки и повышения производительности кода, такие как профилировщики, лучше работают на обычных компьютерах, чем на многопроцессорных.

- *Простое ускорение работы еще ничего не означает.*

  Допустим, ваша параллельная программа работает в 24 раза быстрее на 24-процессорной машине, чем на компьютере с одним процессором. Это просто замечательно, не так ли? Конечно же, если вы всегда получаете линейное повышение скорости и имеете доступ к сколь угодно большому количеству процессоров, то со временем вы перекроете результаты любого последовательного алгоритма. Но тщательно разработанный последовательный алгоритм часто может оказаться эффективнее параллелизируемого кода, выполняемого на типичном многопроцессорном компьютере. Скорее всего, причиной низкой производительности вашей параллельной программы на однопроцессорной машине является недостаточно хороший последовательный алгоритм. Поэтому измерение ускорения работы такой программы при использовании нескольких процессоров является некорректной оценкой преимуществ параллелизма. А приобрести компьютер с неограниченным количеством процессоров — задача не из легких.

  Классическим примером такой ситуации является минимаксный алгоритм поиска в дереве, применяющийся в компьютерных программах игры в шахматы. Поиск методом исчерпывающего перебора в этом дереве удивительно легко поддается параллелизации — просто для каждого поддерева выделяется отдельный процессор. Но при таком подходе большая часть работы этих процессоров выполняется впустую, поскольку одни и те же позиции исследуются на нескольких процессорах. Использование же более интеллектуального алгоритма альфа-бета-отсечений может с легкостью уменьшить объем работы на 99,99%, по сравнению с чем любые достоинства поиска методом исчерпывающего перебора на параллельных процессорах выглядят ничтожными. Алгоритм альфа-бета-отсечений тоже можно запараллелить, но это задача не из легких. При этом увеличение количества используемых процессоров дает неожиданно малое повышение производительности.

- *Параллельные алгоритмы плохо поддаются отладке.*

  За исключением случаев, когда вашу задачу можно разбить на несколько независимых подзадач, при ее исполнении на нескольких процессорах разные процессоры

должны взаимодействовать друг с другом, чтобы выдать правильный конечный результат. К сожалению, вследствие недетерминистического характера этого взаимодействия параллельные программы исключительно трудно отлаживать, поскольку каждое новое исполнение кода будет давать другой результат. Программы с параллельностью по данным обычно не взаимодействуют друг с другом, за исключением копирования результатов в конце исполнения задачи, что значительно упрощает ее выполнение.

Я рекомендую прибегать к параллельным вычислениям только в тех случаях, когда все другие попытки решить задачу последовательным способом оказываются слишком медленными. Но даже в таких случаях я бы ограничился использованием алгоритмов, которые параллелизируют задачу, разбивая ее на несколько подзадач, для исполнения которых разные процессоры не должны общаться друг с другом, кроме как для объединения конечных результатов. Такой примитивный параллелизм может быть достаточно простым как для реализации, так и для отладки, поскольку его конечным результатом, по сути, является хорошая последовательная реализация. Но даже здесь есть свои подводные камни, как показывает следующая история из жизни.

## 5.8. История из жизни. «Торопиться в никуда»

В *разд. 2.9* я описал наши усилия по созданию быстрой программы для исследования проблемы Уоринга. Код был достаточно быстрым и мог решить задачу за несколько недель, работая в фоновом режиме на настольном компьютере. Но моему коллеге этот вариант не понравился.

— Давайте запустим его на многопроцессорном компьютере, — предложил он. — В конце концов, в этом коде внешний цикл выполняет одни и те же вычисления для каждого целого числа от 1 до 1 000 000 000. Я могу разбить этот диапазон чисел на одинаковые интервалы и обработать каждый из них на отдельном процессоре. Разделяй и властвуй. Увидите, как это будет легко.

Он приступил к работе по выполнению нашей программы на суперкомпьютере Intel IPSC-860 с 32 узлами, каждый из которых имел свои собственные 16 Мбайт памяти (очень мощная машина для того времени). Но в течение нескольких следующих недель я постоянно получал от него известия о проблемах. Например:

- программа работает прекрасно, но только прошлой ночью один процессор вышел из строя;
- все шло хорошо, но компьютер случайно перезагрузили, так что мы потеряли все результаты;
- согласно корпоративным правилам никто ни при каких обстоятельствах не может использовать все процессоры более чем 13 часов.

Впрочем, в конце концов, обстоятельства сложились благоприятно. Он дождался стабильной работы компьютера, захватил 16 процессоров (половину компьютера), разбил целые числа от 1 до 1 000 000 000 на 16 одинаковых диапазонов и запустил вычисление каждого диапазона на отдельном процессоре. Следующий день он провел, отбивая атаки разъяренных пользователей, которые не могли попасть на компьютер из-за его

программы. Как только первый процессор закончил обработку своего диапазона (числа от 1 до 62 500 000), он объявил негодующей толпе, что скоро и остальные процессоры закончат свою работу.

Но этого не произошло. Наш коллега не принял во внимание то обстоятельство, что время для обработки каждого целого числа увеличивалось пропорционально увеличению чисел. Оказалось, что проверка представимости числа 1 000 000 000 в виде суммы трех пирамидальных чисел требует больше времени, чем такая же проверка для числа 100. В результате последующие процессоры сообщали об окончании работы через все более длительные интервалы времени. Из-за особенностей устройства суперкомпьютера освободившиеся процессоры нельзя было задействовать снова, пока не была завершена вся работа. В итоге половина машины и большинство ее пользователей оказались в заложниках у одного, последнего процессора.

Какие выводы можно сделать из этой истории? Если вы собираетесь распараллелить вычисления, позаботьтесь о том, чтобы должным образом распределить работу между процессорами. В приведенном примере правильный баланс нагрузки, достигнутый посредством несложных вычислений или с помощью алгоритма разбиения, рассматриваемого в *разд. 10.7*, мог бы значительно сократить время исполнения задачи, а также время, в течение которого нашему коллеге пришлось выносить нападки других пользователей.

## 5.9. Свертка (*)

*Сверткой* двух массивов (или векторов) $A$ и $B$ является новый вектор $C$, для которого

$$C[k]=\sum_{j=0}^{m-1} A[j]\cdot B[k-j].$$

Предположим, что длины $A$ и $B$ равны $m$ и $n$ соответственно и что индексация начинается с 0, тогда естественный диапазон значений для $C$ будет с $C[0]$ по $C[n+m-2]$. Значения всех элементов $A$ и $B$, находящихся вне диапазона, считаются равными нулю, поэтому они не влияют на произведение.

Примером уже знакомой нам свертки является *полиномиальное умножение*. Вспомним задачу умножения двух многочленов, например:

$$(3x^2 + 2x + 6) \times (4x^2 + 3x+2) =$$
$$(3 \cdot 4)x^4 + (3 \cdot 3 + 2 \cdot 4)x^3 + (3 \cdot 2 + 2 \cdot 3 + 6 \cdot 4)x^2 + (2 \cdot 2 + 6 \cdot 3)x^1 + (6 \cdot 2)x^0.$$

Пусть $A[i]$ и $B[i]$ обозначают коэффициенты $x^i$ в каждом из многочленов. Тогда умножение является сверткой, поскольку коэффициент $x^k$-го элемента многочлена произведения выражается приведенной ранее сверткой $C[k]$. Этот коэффициент является суммой произведений всех пар членов, у которых сумма показателей степени равна $k$. Например, $x^5 = x^4 \cdot x^1 = x^3 \cdot x^2$.

Очевидный способ реализации свертки — вычислить скалярное произведение $m$-го члена $C[k]$ для каждого $0 \le k \le n + m - 2$. Для этого потребуются два вложенных цикла, исполняющихся за время $\Theta(nm)$. Вследствие граничных условий во внутреннем цикле не всегда исполняется $m$ итераций. Если бы по бокам $A$ и $B$ находились диапазоны нулевых значений, тогда можно было бы применить более простые границы циклов:

```
for (i = 0; i < n+m-1; i++) {
    for (j = max(0,i-(n-1)); j <= min(m-1,i); j++) {
        c[i] = c[i] + a[j] * b[i-j];
    }
}
```

Свертка умножает все возможные пары элементов из $A$ и $B$, вследствие чего кажется, что для того, чтобы получить эти $n + m - 1$ чисел, должно требоваться квадратическое время. Но подобно чуду сортировки существует хитроумный алгоритм типа «разделяй и властвуй», исполняющийся за время $O(n \log n)$ в предположении, что $n \geq m$. И точно так же, как и сортировка, существует большое количество приложений, которые пользуются этим громадным ускорением для больших последовательностей.

## 5.9.1. Применение свертки

Переход от $O(n^2)$ к $O(n \log n)$ дает свертке такой же большой выигрыш, как и для сортировки. Но чтобы воспользоваться этим, нужно уметь распознавать, когда выполняется операция свертки. Свертки часто возникают при переборе всех возможных способов $k$ выполнения какой-либо операции над диапазонов значений $n$, или же при перемещении маски или шаблона $A$ по последовательности $B$ и выполнении вычислений в каждой позиции.

Далее приводится несколько важных примеров свертки:

- *Умножение целых чисел.*

  Целые числа можно рассматривать как многочлены с любым основанием $b$. Например, $632 = 6 \cdot b^2 + 3 \cdot b^1 + 2 \cdot b^0$, где $b = 10$. Умножение многочленов подобно умножению целых чисел, только без переноса в старшие разряды. Для обработки целых чисел быстрое умножение многочленов можно использовать двумя способами. В первом способе можно явно выполнять операцию переноса на многочлене произведения, добавив $\lfloor C[i]/b \rfloor$ к $C[i+1]$, а затем заменив $C[i]$ на $C[i] \pmod b$. Альтернативно можно вычислить многочлен произведения, а затем определить его значение при $b$, чтобы получить произведение целых чисел $A \times B$.

  При использовании быстрой свертки любой из этих подходов предоставляет даже более быстрый алгоритм умножения, чем алгоритм Карацубы, исполняясь в RAM-модели вычислений за время $O(n \log n)$.

- *Взаимная корреляция.*

  Функция взаимной корреляции измеряет сходство двух временных рядов $A$ и $B$ в виде функции смещения или перестановки одного из них относительно другого. Например, люди покупают продукт в среднем через $k$ дней после просмотра его рекламы. В таком случае между продажами и расходами на рекламу должна быть сильная корреляция с задержкой в $k$ дней.

  Эту функцию взаимной корреляции $C[k]$ можно вычислить по следующей формуле:

$$C[k] = \sum_j A[j]B[j+k].$$

  Обратите внимание, что в этом случае скалярный продукт вычисляется над обратными смещениями $B$, а не над прямыми, как в исходном определении свертки. Но

это произведение все равно можно вычислить при помощи быстрой свертки, просто используя в качестве ввода обратную последовательность $B^R$ вместо прямой $B$.

- *Фильтры скользящего среднего.*

  Нам часто приходится решать задачу сглаживания данных временного ряда, вычисляя среднее значение окна (интервала) значений. Например, нужно вычислить $C[i-1] = 0{,}25B[i-1] + 0{,}5B[i] + 0{,}25B[i+1]$ по всем позициям $i$. Это просто еще одна свертка, где $A$ представляет вектор весовых значений в окне вокруг $B[i]$.

- *Нахождение подстроки в строке.*

  Эту задачу мы рассматривали в *разд. 2.5.3*. В частности, в текстовой строке $S$ нужно было найти все подстроки, совпадающие с шаблоном $P$. Например, подстрока $P = aba$ встречается в тексте $S = abaababa$ в позициях 0, 3 и 5.

  Рассмотренный в *разд. 2.5.3* алгоритм с временной сложностью $O(mn)$ перемещает подстроку шаблона длиной $m$ по всем возможным $n$ начальным точкам текстовой строки поиска. Этот подход с использованием перемещающегося окна напоминает свертку с обратным шаблоном, как показано на рис. 5.5.




**Рис. 5.5.** При изменении образца на обратный свертка строк становится эквивалентной сравниванию строк

Возможно ли решить задачу нахождения подстроки в строке за время $O(n \log n)$ при помощи быстрой свертки? Ответ на этот вопрос является положительным. Предположим, что наши строки созданы из алфавита размером $\alpha$. Тогда каждый символ можно выразить двоичным вектором длиной $\alpha$, содержащим точно один ненулевой бит. Например, для алфавита $\{a, b\}$ буква $a = 10$, а буква $b = 01$. Тогда упомянутые ранее строки $S$ и $P$ можно закодировать при помощи этого вектора следующим образом:

$$S = 1001101001100110$$

$$P = 100110$$

Скалярный продукт элементов в окне будет равен $m$ в четной позиции в $S$ тогда и только тогда, если $P$ начинается в этой позиции в тексте. Таким образом, при помощи быстрой свертки все местонахождения $P$ в $S$ можно найти за время $O(n \log n)$.

#### Подведение итогов

Научитесь распознавать потенциальные свертки. Вашей наградой за обнаружение свертки будет быстрый алгоритм с временем исполнения $\Theta(n \log n)$ вместо $O(n^2)$.

## 5.9.2. Быстрое полиномиальное умножение (**)

Только что рассмотренные приложения должны пробудить у нас интерес к эффективным способам вычисления сверток. В алгоритме быстрой свертки применяется метод «разделяй и властвуй», но подробное доказательство его правильности опирается на

довольно сложные свойства комплексных чисел и линейную алгебру, которые выходят за рамки рассматриваемого в этой книге материала. *Поэтому при желании вы можете свободно пропустить это рассмотрение.* Но я предоставлю вам достаточно вводной информации, чтобы можно было понять, при чем здесь «разделяй и властвуй».

Свертка представляется посредством быстрого алгоритма для умножения многочленов, основанном на ряде наблюдений:

- *Многочлены можно представать или в виде уравнений, или в виде множеств точек.*

  Как мы знаем, каждая пара точек определяет линию. В более широком смысле любой многочлен $P(x)$ степени $n$ полностью определяется посредством $n + 1$ точек по многочлену. Например, точки $(-1, -2)$, $(0, -1)$ и $(1, 2)$ определяют квадратное уравнение $y = x^2 + 2x - 1$ (а также определяются этим уравнением).

- *Для многочлена $P(x)$ эти $n + 1$ точек можно вычислить, но это выглядит затратным.*

  Создание точки по заданному многочлену не представляет никакого труда. Для этого просто выбираем произвольное значение $x$ и вставляем его в $P(x)$. Время для обработки одного такого значения $x$ будет линейным со степенью $P(x)$, что означает $n$ для интересующих нас задач. Но выполнение этой операции $n + 1$ раз для разных значений $x$ займет время $O(n^2)$, что слишком много, если мы хотим быстрого умножения.

- *Умножить многочлены $A$ и $B$ в точечном представлении будет легко, если они оба вычислены с одинаковыми значениями $x$.*

  Предположим, что нужно вычислить произведение $(3x^2 + 2x + 6)(4x^2 + 3x + 2)$. Результатом будет многочлен четвертой степени, поэтому для его определения потребуется пять точек. Для вычисления обоих множителей можно использовать одинаковые значения $x$:

$$A(x) = 3x^2 + 2x + 6 \longrightarrow (-2, 14), (-1, 7), (0, 6), (1, 11), (2, 22),$$
$$B(x) = 4x^2 + 3x + 2 \longrightarrow (-2, 12), (-1, 3), (0, 2), (1, 9), (2, 24).$$

  Так как $C(x) = A(x)B(x)$, то точки на $C(x)$ можно создать, умножая соответствующие значения $y$:

$$C(x) \leftarrow (-2, 168), (-1, 21), (0, 12), (1, 99), (2, 528).$$

  Таким образом, умножение точек в этом представлении занимает только линейное время.

- *Многочлен $A(x)$ $n$-й степени можно вычислить как два многочлена степени $n/2$ в $x^2$.*

  Члены многочлена $A$ можно разделить на группы с четной и нечетной степенью, например

$$12x^4 + 17x^3 + 36x^2 + 22x + 12 = (12x^3 + 36x^2 + 12) + x(17x^2 + 22).$$

  Заменив $x^2$ на $x'$, получим в правой части два меньших многочлена более низкой степени, как и обещалось.

- *Все это наводит на мысль об эффективном алгоритме типа «разделяй и властвуй».*

  Нам нужно вычислить $n$ точек многочлена $d$-й степени. Поскольку мы будем использовать эти точки для вычисления произведения двух многочленов, то нам нужно $n \geq 2d + 1$ точек. Эту задачу можно разложить на две части: выполнение такого вычисления на двух многочленах половинной степени плюс объединение результатов за линейное время. Таким образом определяется рекуррентное соотношение $T(n) = 2T(n/2) + O(n)$, которое дает $O(n \log n)$.

- *Чтобы все это работало должным образом, нужно выбрать правильные значения x для вычислений.*

  Свойство операции возведения в квадратную степень делает желательным использовать пары выбранных точек в виде $\pm x$, т. к. для их обработки требуется наполовину меньше времени, поскольку они дают одинаковые значения после возведения в квадрат.

  Но это свойство не сохраняется при рекурсии, если только для значений $x$ не используются правильно подобранные комплексные числа. Множеством решений уравнения $x^n = 1$ являются *корни n-й степени из единицы*. Для реальных чисел можно получить только решение $x \in \{-1, 1\}$, но для комплексных чисел существует $n$ решений. Корень $k$ из этих $n$ корней определяется следующим уравнением:

$$w_k = \cos(2k\pi/n) + i \sin(2k\pi/n).$$

  Чтобы оценить волшебные свойства этих чисел, посмотрим, что происходит при возведении их в степень:

$$w = \left\{1, \frac{1+i}{\sqrt{2}}, i, -\frac{1-i}{\sqrt{2}}, -1, \frac{1+i}{\sqrt{2}}, -i, \frac{1-i}{\sqrt{2}}\right\},$$

$$w^2 = \{1, i, -1, -i, 1, i, -1, -i\},$$

$$w^4 = \{1, -1, 1, -1, 1, -1, 1, -1\},$$

$$w^8 = \{1, 1, 1, 1, 1, 1, 1, 1\}.$$

  Обратите внимание на то, что эти члены представлены в виде пар положительного и отрицательного элементов, а количество отдельных членов уменьшается наполовину с каждым возведением в квадрат. Вот эти свойства должны присутствовать, чтобы можно было применять метод типа «разделяй и властвуй».

Наилучшие реализации быстрой свертки обычно вычисляют быстрое преобразование Фурье (БПФ), поэтому обычно мы стремимся свести задачу к этой форме, чтобы воспользоваться существующими библиотеками. Быстрые преобразования Фурье рассматриваются в *разд. 16.11*.

Быстрая свертка решает многие важные задачи за время $O(n \log n)$. Первый шаг к решению таких задач методом свертки — распознать, что они поддаются этому решению.

## Замечания к главе

Более подробное обсуждение алгоритмов типа «разделяй и властвуй» можно найти в книгах, [KT06] и [Man89]. Отличный обзор основной теоремы дается в книге [CLRS09].

Доступное введение в область алгоритмической разработки вакцин приведено в книге [Ski12]. Последовательности поиска ошибки, описанные в *разд. 5.2*, являются примером проекта для работы с коллективными данными, позволяющего идентифицировать, например, одного больного пациента среди $n$ пациентов, выполнив только $\lg n$ анализов по пулу проб крови. Исследование теории этих интересных проектов приводится в книге [DH00]. Упорядоченность слева направо подмножеств в этих проектах отражает код Грея, в котором разница между смежными подмножествами составляет ровно один элемент. Коды Грея рассматриваются в *разд. 17.5*.

Наша работа по параллельным вычислениям с пирамидальными числами была представлена в журнале [DY94]. Мое рассмотрение сверток и БПФ было основано на заметках по лекциям об алгоритмах 15-451/651 Аврима Блюма (Avrim Blum).

## 5.10. Упражнения

### Двоичный поиск

1. *[3]* Допустим, имеется упорядоченный массив $A$ из $n$ чисел, который был *циклически сдвинут* вправо на $k$ позиций. Например, последовательность {35, 42, 5, 15, 27, 29} представляет собой отсортированный массив, циклически сдвинутый вправо на $k = 2$ позиций, а последовательность {27, 29, 35, 42, 5, 15} — массив, циклически сдвинутый на $k = 4$ позиций.

   a) Допустим, что значение $k$ известно. Разработайте алгоритм с временем исполнения $O(1)$ для поиска наибольшего числа в массиве $A$.

   b) Допустим, что значение $k$ неизвестно. Разработайте алгоритм с временем исполнения $O(\lg n)$ для поиска наибольшего числа в массиве $A$. Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $O(n)$.

2. *[3]* Упорядоченный массив размером $n$ содержит различные целые числа от 1 до $n + 1$, при этом одно число отсутствует. Разработайте алгоритм с временной сложностью $O(\log n)$, чтобы определить отсутствующее целое число, не прибегая к использованию дополнительного места.

3. *[3]* В игре «20 вопросов» первый игрок задумывает число от 1 до $n$. Второй игрок должен угадать это число, задав как можно меньше вопросов, требующих ответа «да» или «нет». Предполагается, что оба играют честно.

   a) Какой должна быть оптимальная стратегия второго игрока, если число $n$ известно?

   b) Какую стратегию можно применить, если число $n$ неизвестно?

4. *[5]* Имеется *одномодальный* массив, содержащий $n$ различных элементов. В одномодальном массиве элементы сначала упорядочены в возрастающем порядке, а затем в убывающем. Разработайте алгоритм с временной сложностью $O(\log n)$ для нахождения максимального элемента массива.

5. *[5]* Дана отсортированная последовательность $[a_1, a_2, ..., a_n]$ *разных* целых чисел. Разработайте алгоритм с временем исполнения $O(\lg n)$ для определения, содержит ли массив такой индекс $i$, для которого $a_i = i$. Например, в массиве $[-10, -3, 3, 5, 7]$, $a_3 = 3$. Но массив $[2, 3, 4, 5, 6, 7]$ не имеет такого индекса $i$.

6. *[5]* Дана отсортированная последовательность $a = [a_1, a_2, ..., a_n]$ *разных* целых чисел в диапазоне от 1 до $m$, где $n < m$. Разработайте алгоритм с временем исполнения $O(\lg n)$ для поиска целого числа $x \leq m$, отсутствующего в этой последовательности. Чтобы решение было засчитано полностью, алгоритм должен находить наименьшее такое целое число $1 \leq x \leq m$.

7. *[5]* Пусть $M$ — матрица размером $n \times m$, в которой элементы каждой строки отсортированы в возрастающем порядке слева направо, а элементы каждого столбца отсортированы в возрастающем порядке сверху вниз. Разработайте эффективный алгоритм для определения местонахождения целого числа $x$ в матрице $M$ или для определения, что матрица не содержит это число. Сколько сравнений числа $x$ с элементами матрицы выполняет ваш алгоритм в наихудшем случае?

## Алгоритмы типа «разделяй и властвуй»

8. *[5]* Дано два отсортированных массива $A$ и $B$ размером $n$ и $m$ соответственно. Разработайте алгоритм с общей временной сложностью $O(\log(m + n))$ для нахождения медианы элементов $n + m$.

9. *[8]* Задача нахождения поддиапазона с наибольшей суммой значений для входного массива $A$ из $n$ значений (рассматривается в *разд. 5.6*) заключается в нахождении пары таких значений индекса $i$ и $j$, которые максимизируют значение $S = \sum_{k=1}^{j} A[k]$. Разработайте алгоритм с временной сложностью $O(n)$ для решения этой задачи.

10. *[8]* Имеется $n$ деревянных палочек, длина каждой из которых равна определенному целому числу, и длина $i$-й палочки равна $L[i]$. Требуется разрезать эти палочки таким образом, чтобы получить $k$ палочек абсолютно одинаковой длины, кроме оставшихся фрагментов. Более того, эти $k$ палочек должны быть максимально возможной длины.

    a) Какой будет конечная максимальная длина $k = 4$ палочек, исходная длина которых равна L = {10, 6, 5, 3}? (Подсказка: это не 3.)

    b) Разработайте *правильный* и эффективный алгоритм, который для данных $L$ и $k$ возвращает максимально возможную длину $k$ одинаковых палочек, полученных из исходных $n$ палочек.

11. *[8]* Расширьте алгоритм сравнения строк на основе свертки, рассмотренный ранее в тексте главы, для работы с символами подстановки *, которые соответствуют любому символу. Например, искомая строка *sh*t* должна совпадать со строками *shot* и *shut*.

## Рекуррентные соотношения

12. *[5]* В *разд. 5.3* утверждается, что любой многочлен можно выразить в виде рекуррентного соотношения. Определите рекуррентное соотношение, представляющее многочлен $a_n = n^2$.

13. *[5]* Имеется выбор следующих трех алгоритмов:
    - алгоритм $A$ решает задачи, разделяя их на пять подзадач половинного размера, рекурсивно решая каждую из подзадач, а затем соединяя полученные решения за линейное время;
    - алгоритм $B$ решает задачи размером $n$, рекурсивно решая две подзадачи размером $n - 1$, а затем соединяя полученные решения за постоянное (константное) время;
    - алгоритм $C$ решает задачи размером $n$, разделяя их на девять подзадач размером $n/3$, рекурсивно решая каждую из подзадач, а затем соединяя полученные решения за время $\Theta(n^2)$.

    Каково время исполнения каждого из этих алгоритмов (используя обозначение $O$-большое) и какой из них вы бы выбрали?

14. *[5]* Решите следующие рекуррентные соотношения и укажите границу $\Theta$ для каждого из них:

    a) $T(n) = 2T(n/3) + 1$;

    b) $T(n) = 5T(n/4) + n$;

    c) $T(n) = 7T(n/7) + n$;

    d) $T(n) = 9T(n/3) + n^2$.

15. *[3]* Решите следующие рекуррентные соотношения, используя основную теорему:

    a) $T(n) = 64T(n/4) + n^4$;

    b) $T(n) = 64T(n/4) + n^3$;

    c) $T(n) = 64T(n/4) + 128$.

16. *[3]* Укажите асимптотически жесткие верхние границы ($O$-большое) для $T(n)$ в каждом из следующих рекуррентных соотношений. Обоснуйте свои решения, указав конкретный случай основной теоремы, итерируя рекурсию или используя метод подстановки:

    a) $T(n) = T(n - 2) + 1$;

    b) $T(n) = 2T(n/2) + n \lg^2 n$;

    c) $T(n) = 9T(n/4) + n^2$.

## LeetCode

1. **https://leetcode.com/problems/median-of-two-sorted-arrays/**
2. **https://leetcode.com/problems/count-of-range-sum/**
3. **https://leetcode.com/problems/maximum-subarray/**

## HackerRank

1. **https://www.hackerrank.com/challenges/unique-divide-and-conquer**
2. **https://www.hackerrank.com/challenges/kingdom-division/**
3. **https://www.hackerrank.com/challenges/repeat-k-sums/**

## Задачи по программированию

Эти задачи доступны на сайте **https://.onlinejudge.org**:

1. «Polynomial Coefficients», глава 5, задача 10105.
2. «Counting», глава 6, задача 10198.
3. «Closest Pair Problem», глава 14, задача 10245.

# Хеширование и рандомизированные алгоритмы

Большинство рассмотренных в предыдущих главах алгоритмов были разработаны с целью оптимизация производительности в наихудшем случае — они гарантированно возвращают оптимальное решение для каждого экземпляра задачи в рамках указанного времени исполнения.

Это прекрасно, когда это возможно. Но ослабив требование, чтобы алгоритм был или всегда правильный, или всегда эффективный, можно получить полезные алгоритмы, которые продолжают предоставлять гарантированную производительность. Рандомизированные алгоритмы — это не просто эвристические алгоритмы: любой случай низкой производительности обусловливается невезением при бросках монеты, а не конфликтными входными данными.

В зависимости от предоставляемой ими гарантии: правильности или эффективности — рандомизированные алгоритмы делятся на следующие две категории:

- *Алгоритмы типа Лас-Вегас.*
  Эти рандомизированные алгоритмы гарантируют правильность и обычно (но не всегда) выдают хорошую эффективность. Отличным примером алгоритма Лас-Вегас будет алгоритм быстрой сортировки (quicksort).
- *Алгоритмы типа Монте-Карло.*

  Эти алгоритмы доказуемо эффективные и обычно (но не всегда) предоставляют правильный результат или нечто близкое к нему. Типичными алгоритмами этой категории являются методы произвольной выборки (см. *разд. 12.6.1*), которые возвращают наилучшее решение, полученное в результате, скажем, 1 000 000 произвольных выборок.

В этой главе мы рассмотрим несколько примеров алгоритмов каждой из этих категорий.

Одно из положительных качеств рандомизированных алгоритмов это то, что они очень легко поддаются описанию и реализации. Устранение необходимости беспокоиться о редких или маловероятных ситуациях позволяет избежать сложных структур данных или иных нестандартных ситуаций. Эти аккуратные рандомизированные алгоритмы часто интуитивно привлекательны и сравнительно легко поддаются разработке.

Но, с другой стороны, рандомизированные алгоритмы часто весьма трудно тщательно анализировать. Для анализа рандомизированных алгоритмов необходимо применять математику теории вероятностей, которая в силу необходимости является как формальной, так и утонченной. Вероятностный анализ часто сводится к алгебраическому манипулированию длинными цепочками неравенств страшного вида и полагается на использование разных приемов и опыт.

По этой причине будет трудно предоставить их удовлетворительный анализ на уровне этой книги, в которой мы придерживаемся строгой политики ухода от доказательства теорем. Но я попытаюсь предоставить объяснения на интуитивном уровне, чтобы вы могли оценить, почему эти алгоритмы обычно правильные или эффективные.

Мы вкратце взглянули на рандомизированные алгоритмы при рассмотрении таблиц хеширования в *разд. 3.7* и быстрой сортировки в *разд. 4.6*. Вернитесь сейчас к материалу этих разделов, чтобы дать себе наилучший шанс разобраться с последующим материалом.

**ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Город быстрой сортировки**

**ЗАДАЧА**. Почему рандомизированный алгоритм быстрой сортировки является алгоритмом Лас-Вегас, а не алгоритмом Монте-Карло?

**РЕШЕНИЕ**. Согласно приведенной ранее категоризации алгоритмы Монте-Карло всегда быстрые и обычно правильные, тогда как алгоритмы Лас-Вегас всегда правильные и обычно быстрые.

Рандомизированный алгоритм быстрого поиска всегда создает упорядоченную перестановку, поэтому мы знаем, что он всегда правильный. Вследствие выбора очень плохого ряда опорных точек время исполнения может превысить $O(n \log n)$, но в конечном итоге сортировка всегда будет выполнена. Таким образом, алгоритм быстрой сортировки (quicksort) является отличным примером алгоритма категории Лас-Вегас. ■

# 6.1. Обзор теории вероятностей

В рамках моего намерения удерживать объем этой книги в разумных пределах, я буду сопротивляться соблазну предоставить здесь исчерпывающий обзор теории вероятностей. Но при этом я предполагаю следующее:

- вам уже приходилось в прошлом соприкасаться с теорией вероятностей;
- вы знаете, где в случае надобности искать дополнительную информацию.

Поэтому я ограничусь рассмотрением нескольких базовых определений и свойств, с которыми мы будем работать.

## 6.1.1. Теория вероятностей

Теория вероятностей предоставляет формальную основу для логических рассуждений о вероятности событий. Поскольку это формальная дисциплина, с ней связано множество определений, конкретизирующих, о чем именно ведутся рассуждения:

- *Эксперимент* — это процедура, создающая один из множества возможных результатов. В качестве примера эксперимента можно привести бросание двух шестигранных игральных костей. Одна из костей — красная, другая — синяя, и на каждой грани каждой кости нанесено отдельное целое число в диапазоне $\{1, ..., 6\}$.
- *Пространство выборок* $S$ — это множество возможных результатов эксперимента. В нашем примере с игральными костями возможны 36 результатов (рис. 6.1).

$$S = \{(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),$$
$$(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),$$
$$(3,1),(3,2),(3,3),(3,4),(3,5),(3,6),$$
$$(4,1),(4,2),(4,3),(4,4),(4,5),(4,6),$$
$$(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),$$
$$(6,1),(6,2),(6,3),(6,4),(6,5),(6,6)\}.$$

**Рис. 6.1.** 36 результатов бросания двух шестигранных игральных костей (левая цифра — красная кость, правая цифра — синяя кость)

- *Событие E* — это обусловленное подмножество результатов эксперимента. Например, событие, при котором сумма значений костей будет равной 7 или 11 (условие выигрыша на первом броске в игре крэпс), выражается следующим подмножеством (рис. 6.2).

$$E = \{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1),(5,6),(6,5)\}$$

**Рис. 6.2.** Подмножество событий, при которых сумма значений костей будет равной 7 или 11 (левая цифра — красная кость, правая цифра — синяя кость)

- *Вероятность результата s*, обозначаемая $p(s)$, является числом с двумя свойствами:
  - для каждого результата $s$ в пространстве выборок S: $0 \le p(s) \le 1$;
  - сумма вероятностей всех результатов равна единице:

$$\sum_{s \in S} p(s) = 1.$$

  При условии двух различных правильных костей для всех $s \in S$ вероятность

$$p(s) = (1/6) \times (1/6) = 1/36.$$

- *Вероятность события E* равна сумме вероятностей результатов события. Таким образом,

$$P(E) = \sum_{s \in E} p(s).$$

  Альтернативно это определение может быть использовано с дополнением события $\overline{E}$ — для случая, когда событие $E$ не происходит. Тогда

$$P(E) = 1 - P(\overline{E}).$$

  Это полезное определение, поскольку часто более удобно анализировать $P(\overline{E})$, чем непосредственно $P(E)$

- *Произвольная переменная V* является числовой функцией на результатах пространства вероятностей. Эта функция «сумма значений двух костей» ($V((a, b)) = a + b$) создает целочисленный результат от 2 до 12. Он неявно выражает распределение вероятностей возможных значений этой произвольной переменной. Например, вероятность $P(V(s) = 7) = 1/6$, тогда как $P(V(s) = 12) = 1/36$.

- *Ожидаемое значение* произвольной переменной $V$, определенной на пространстве выборок $S$, $E(V)$, определяется следующим образом:

$$E(V) = \sum_{s \in S} p(s) \cdot V(s).$$

## 6.1.2. Составные события и независимость

Для нас будут представлять интерес составные события, вычисляемые из более простых событий $A$ и $B$ на одном и том же множестве результатов. Возможно, событие $A$ состоит в том, что как минимум на одной из двух костей выпадет четное число, тогда как событие $B$ будет заключаться в сумме броска, равной 7 или 11. Обратите внимание на существование результатов $A$, не являющимися результатами $B$, — такой вариант показан на рис. 6.3.

$$A - B = \{(1, 2), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), (2, 6), (3, 2), (3, 6), (4, 1),$$
$$(4, 2), (4, 4), (4, 5), (4, 6), (5, 4), (6, 2), (6, 3), (6, 4), (6, 6)\}$$

**Рис. 6.3.** Результаты события $A$, не являющиеся результатами события $B$ (левая цифра — красная кость, правая цифра — синяя кость)

Эта операция называется *разностью множеств*. Также обратите внимание на то, что в нашем случае $B - A = \{\}$, поскольку каждая пара чисел, сумма которых составляет 7 или 11, должна содержать одно нечетное и одно четное число.

Общие результаты для обоих событий $A$ и $B$ называются *пересечением* и обозначаются $A \cap B$. Это можно записать следующим образом:

$$A \cap B = A - (S - B).$$

Результаты, содержащиеся или в A, или в B, называются *объединением* этих множеств, и обозначаются $A \cup B$. Вероятности объединения и пересечения связаны следующей формулой:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Добавив к приведенным здесь операциям операцию *разности*: $\overline{A} = S - A$, получим мощный язык для комбинирования событий (рис. 6.4). Суммируя вероятности результатов множеств, можно легко вычислить вероятность любого из этих множеств.

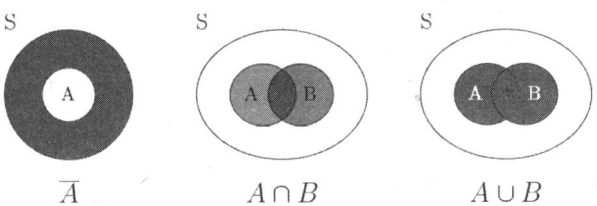



**Рис. 6.4.** Диаграммы Венна, иллюстрирующие разность (*слева*), пересечение (*в центре*) и объединение (*справа*) множеств

События $A$ и $B$ называются *независимыми*, если

$$P(A \cap B) = P(A) \times P(B).$$

Это означает, что события $A$ и $B$ не имеют специальной структуры общих результатов. При этом, предполагая, например, что в классе половина учащихся женского пола, и что успеваемость половины учащихся в этом же классе выше средней, при независимости этих событий можно ожидать, что четверть учащихся класса — это учащиеся женского пола с успеваемостью выше средней. Теоретики в области теории вероятностей обожают независимые события, т. к. они упрощают вычисления. Например, если $A_i$ обозначает событие получения четного числа на $i$-м броске одной игральной кости, то вероятность получить все четные числа при броске двух игральных костей равна

$$P(A_i \cap A_2) = P(A_1)P(A_2) = (1/2)(1/2) = 1/4.$$

Тогда вероятность события $A$, при котором как минимум число на одной из костей будет четным, определяется следующей формулой:

$$P(A) = P(A_1 \cup A_2) = P(A_1) + P(A_2) - P(A_1 \cap A_2) = 1/2 + 1/2 - 1/4 = 3/4.$$

Но независимость часто не выдерживается, что объясняет большую часть сложности и трудности вероятностного анализа. Вероятность выпадения $n$-го количества «орлов» при броске $n$ независимых монет равна $1/2^n$. Но если бы монеты были идеально коррелированные, тогда вероятность этого была бы 1/2, поскольку в таком случае единственными возможными результатами были бы или все «орлы», или все «решки». При наличии сложных зависимостей между результатами бросания $i$-й и $j$-й монет это вычисление становится очень трудным.

Рандомизированные алгоритмы обычно разрабатываются на основе независимо выбираемых случайным образом образцов, чтобы можно было безопасно умножать вероятности для понимания составных событий.

## 6.1.3. Условная вероятность

Предполагая, что $P(B) > 0$, *условная вероятность* $A$ при $B$, т. е. $P(A|B)$, определяется следующим образом:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}.$$

В частности, если события $A$ и $B$ независимы, тогда

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A)$$

и событие $B$ никоим образом не влияет на вероятность события $A$. Условная вероятность представляет интерес только в том случае, когда два события зависят друг от друга.

Вспомним события бросания игральных костей из *разд. 6.1.2*. А именно:

- событие $A$ — число как минимум на одной из костей четное;
- событие $B$ — сумма чисел обеих костей или 7, или 11.

Обратите внимание на то, что $P(A|B) = 1$, т. к. при *любом* броске с нечетной сумой чисел одно из чисел должно быть четным, а другое — нечетным. Таким образом, $A \cap B = B$. Обратите внимание на то, что для $P(B|A)$: $P(A \cap B) = P(B) = 8/36$ и $P(A) = 27/36$, поэтому $P(B|A) = 8/27$.

Для вычисления условных вероятностей нашим основным инструментом будет теорема Байеса, которая изменяет на обратное направление зависимостей:

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}.$$

Часто, как и в случае с этой задачей, вычислить вероятности в одном направлении более трудно, чем в другом. Применяя теорему Байеса, получаем

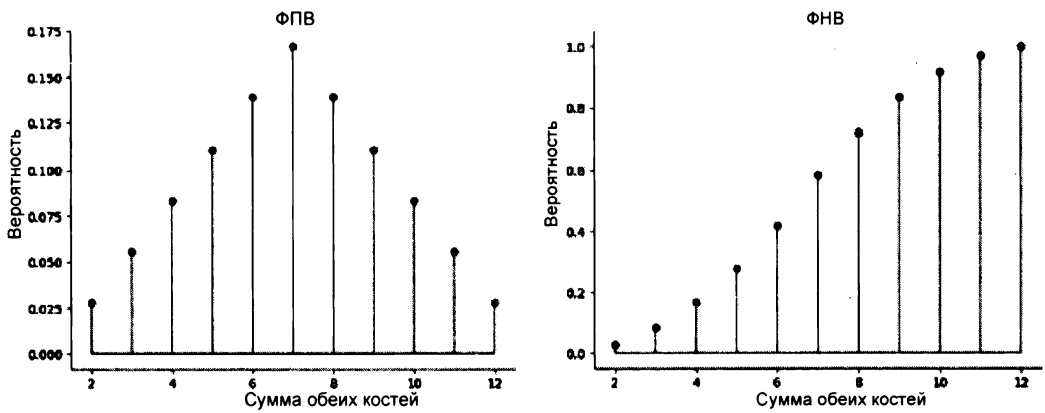
$$P(B|A) = (1 \cdot 8/36)/(27/36) = 8/27,$$

т. е. точно такой же результат, как и раньше.

## 6.1.4. Распределения вероятностей

Случайные переменные — это числовые функции, в которых значения связаны с вероятностями проявления. В нашем примере, где $V(s)$ — это сумма чисел на двух костях, функция выдает целое число от 2 до 12. Вероятность определенного значения $V(s) = X$ равна сумме вероятностей всех результатов, сумма составляющих которых равна $X$.

Такие случайные переменные можно представить их *функцией плотности вероятностей* (ФПВ). Эту функцию отображает график, в котором ось $x$ представляет значения, которые может принимать случайная переменная, а ось $y$ — вероятность каждого значения. На рис. 6.5, *а* приводится ФПВ суммы двух правильных игральных костей. Обратите внимание, что пик в точке $X = 7$ соответствует наиболее вероятной сумме костей, вероятность которой равняется 1/6.




*а* *б*

**Рис. 6.5.** Функция плотности вероятностей (ФПВ) (*а*) суммы двух костей содержит точно такую же информацию, что и функция накопленных вероятностей (ФНВ) (*б*), но выглядит совершенно иначе

## 6.1.5. Среднее и дисперсия

Существуют два основных типа сводных статистических данных, которые совместно предоставляют нам громадный объем информации о распределении вероятностей или множестве данных:

- *измерения средних значений* определяют середину, вокруг которой распределены произвольные выборки или точки данных;
- *дисперсия*, или *измерения изменчивости*, описывает разброс произвольных выборок или точек от центра.

Основной мерой центрированности является *среднее значение*. Среднее значение произвольной переменной $V$, обозначаемое как $E(V)$, также называется *ожидаемым значением* и вычисляется по следующей формуле:

$$E(V) = \sum_{s \in S} V(s)p(s).$$

Когда все элементарные события имеют одинаковую вероятность, для вычисления среднего значения используется следующая формула:

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

Наиболее распространенная мера изменчивости — *стандартное отклонение* σ. Стандартное отклонение случайной переменной $V$ вычисляется по формуле

$$\sigma = \sqrt{E\left((V - E(V))\right)^2}.$$

Для множества данных стандартное отклонение вычисляется по сумме квадратов разниц между отдельными элементами и средним значением:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \overline{X}\right)^2}{n-1}}.$$

Связанная статистическая характеристика *дисперсия* вычисляется как квадрат стандартного отклонения $V = \sigma^2$. Иногда в обсуждении более удобно использовать дисперсию, чем стандартное отклонение, поскольку этот термин больше чем вдвое короче. Но обе они измеряют абсолютно одинаковую характеристику.

## 6.1.6. Броски монет

Вы, скорее всего, можете довольно аккуратно определить на интуитивном уровне распределение количеств «орлов» и «решек» в 10 000 бросков правильной монеты. Вы знаете, что в $n$ бросках, в каждом из которых вероятность выпадения «орла» составляет $p = 1/2$, ожидаемое количество «орлов» равно произведению $pn$, или 5000 в нашем примере. Также вы, скорее всего, знаете, что распределение для $h$ «орлов» из $n$ бросков является биноминальным распределением, где

$$P(X - h) = \frac{\binom{n}{h}}{\sum_{x=0}^{n}\binom{n}{x}} = \frac{\binom{n}{h}}{2^n}.$$

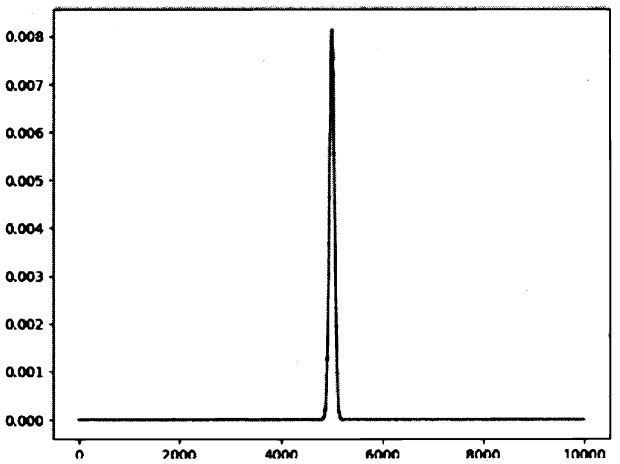



**Рис. 6.6.** Распределение вероятности выпадения $h$ «орлов» при $n$ бросках правильной монеты плотно отцентрировано вокруг среднего значения и равно $n/2$. В нашем примере для $n = 10\,000$ распределение $h = n/2 = 5000$

А это симметричное распределение в виде колокола вокруг среднего значения. Но вы можете не догадываться, насколько узкое это распределение (рис. 6.6).

Конечно же, теоретически, при $n$ бросках правильной монеты количество «орлов» *может* быть каким угодно в диапазоне от 0 до $n$. Но оно таким не будет. Количество выпавших «орлов» будет *почти всегда* в пределах нескольких стандартных отклонений от среднего, где стандартное отклонение $\sigma$ для биноминального распределения определяется следующей формулой

$$\sigma = \sqrt{np(1-p)} = \Theta\left(\sqrt{n}\right).$$

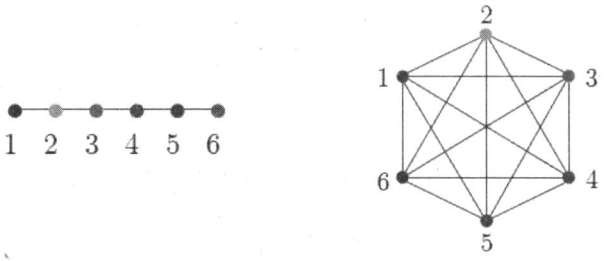
В самом деле, для любого распределения вероятностей как минимум $1 - (1/k^2)$-я часть массы распределения находится в пределах $\pm k\sigma$ от среднего значения $\mu$. Обычно стандартное отклонение $\sigma$ невелико — сравнительно с $\mu$ для распределений, возникающих при анализе рандомизированных алгоритмов и процесса.

#### Подведение итогов

Студенты часто спрашивают меня, что будет, когда рандомизированный быстрый поиск исполняется за время $\Theta(n^2)$. Ничего не будет, точно так же, как ничего не будет при покупке лотерейного билета — вы практически наверняка просто ничего не выиграете. А с рандомизированным быстрым поиском вы практически наверняка просто выиграете, поскольку его распределение вероятности настолько плотное, что его время исполнения почти всегда близко к ожидаемому.

## Остановка для размышлений: Случайный обход графа

**ЗАДАЧА.** Понимание процесса случайного обхода графа весьма важно. Ожидаемое время обхода (количества ходов, требуемых для посещения всех вершин) будет разным в зависимости от топологии графа (рис. 6.7). В частности, сколько займет прохождение маршрута (рис. 6.7, *а*)?




*а* *б*

**Рис. 6.7.** Сколько займет проход случайным образом по всем $n$ вершинам маршрута (*а*) и графа (*б*)?

Предположим, что обход начинается с левого конца маршрута, содержащего $m$ вершин. Направление перемещения определяется броском правильной монеты. Если выпадает «орел», делаем один ход вправо, а если «решка» — один ход влево. Если ход выводит нас за пределы маршрута, стоим на месте. Сколько бросков монеты как функции $m$ потребуется, чтобы дойти до правого конца маршрута?

**РЕШЕНИЕ**. Чтобы добраться до правого конца маршрута, после $n$ бросков количество выброшенных «орлов» должно быть на $m - 1$ больше, чем «решек», — и это в предположении, что броски не выполняются, когда мы находимся в левом конце маршрута, откуда можно перемещаться только вправо. Ожидается, что приблизительно в половине бросков выпадет «орел» со стандартным отклонением, равным $\sigma = \Theta\left(\sqrt{n}\right)$. Это отклонение $\sigma$ описывает размах вероятной разницы между количествами «орлов» и «решек». Чтобы получить значение $\sigma$ в районе значения $m$, нам нужно выполнить достаточное количество бросков. Поэтому

$$m = \Theta\left(\sqrt{n}\right) \rightarrow n = \Theta\left(m^2\right).$$

■

# 6.2. Задача мячиков и контейнеров

Задача мячиков и контейнеров — классическая задача теории вероятностей. Задача заключается в произвольном бросании $x$ одинаковых мячиков в $y$ помеченных контейнеров. Нас интересует полученное распределение мячиков в контейнерах. Сколько контейнеров будут содержать то или иное количество мячиков?

Хеширование можно рассматривать как процесс бросания мячиков в контейнеры. Предположим, что имеется $n$ мячиков/ключей и $n$ контейнеров/корзин. Ожидается, что в среднем каждый контейнер будет содержать по одному мячику. Но обратите внимание на то, что такое распределение будет выдерживаться *независимо* от качества функции хеширования.

Хорошая функция хеширования должна работать подобно генератору случайных чисел, выбирая идентификаторы контейнеров/целых чисел с одинаковой вероятностью от 1 до $n$. Но что будет при выборе $n$ таких целых чисел из равномерного распределения? Идеальным результатом было бы, чтобы каждый из $n$ объектов (мячиков) попадал

в отдельный контейнер, чтобы в каждой корзине было точно по одному объекту для поиска. Но действительно ли это происходит на самом деле?

В качестве упражнения для развития интуиции я рекомендую вам написать небольшую программу эмулирования этой задачи и провести эксперимент. Моя программа выдала следующие результаты для таблиц хеширования, содержащих от одного миллиона до ста миллионов объектов (рис. 6.8).

| | Количество корзин, содержащих $k$ объектов | | |
|---|---|---|---|
| $k$ | $n = 10^6$ | $n = 10^7$ | $n = 10^8$ |
| 0 | 367,899 | 3,678,774 | 36,789,634 |
| 1 | 367,928 | 3,677,993 | 36,785,705 |
| 2 | 183,926 | 1,840,437 | 18,392,948 |
| 3 | 61,112 | 613,564 | 6,133,955 |
| 4 | 15,438 | 152,713 | 1,531,360 |
| 5 | 3130 | 30,517 | 306,819 |
| 6 | 499 | 5,133 | 51,238 |
| 7 | 56 | 754 | 7,269 |
| 8 | 12 | 107 | 972 |
| 9 | | 8 | 89 |
| 10 | | | 10 |
| 11 | | | 1 |

**Рис. 6.8.** Результаты для таблиц хеширования, содержащих от одного миллиона до ста миллионов объектов

Можно видеть, что во всех трех случаях 36,78% всех корзин остаются пустыми. Это не может быть простым совпадением. Первая корзина будет пустой тогда и только тогда, когда каждый из $n$ мячиков попадет в одну из других $n - 1$ корзин. Вероятность $p$ непопадания для каждого конкретного мячика равна $p = (n\ 1)/n$ и стремится к 1 по мере возрастания $n$. Но не попасть должны *все* $n$ мячиков, и вероятность такого события равна $p^n$. Что же будет, если умножить большое количество больших вероятностей? По сути, мы видели ответ на этот вопрос ранее при изучении пределов:

$$P(|B_1| = 0) = \left(\frac{n-1}{n}\right)^2 \to \frac{1}{e} = 0{,}367879.$$

Таким образом, 36,78% корзин в большой таблице хеширования будут пустыми. Кроме этого, как оказывается, ожидается, что точно такая же доля корзин будет содержать по одному объекту.

Если такое большое количество корзин пустует, то другие корзины должны быть переполненными. В приведенной на рис. 6.8 таблице можно видеть, что с увеличением $n$ количество объектов в самой заполненной корзине увеличивается от 8 до 9 и до 11. В действительности, ожидаемое значение наиболее длинного списка равно $O(\log n/\log \log n)$ и медленно нарастает, т. е. не является константным (постоянным). Поэтому я несколько поторопился, когда в *разд. 3.7.1* сказал, что в наихудшем случае время доступа для хеширования равно $O(1)$[1].

[1] Точнее говоря, ожидаемое среднее время поиска для всех $n$ ключей действительно будет $O(1)$, но при этом следует ожидать несколько достаточно неудачных ключей, требующих время $\Theta(\log n/\log \log n)$.

### Подведение итогов

Для точного анализа случайного процесса требуются знания формальной теории вероятностей, алгебры и тщательная асимптотика. В этой главе указанные предметы рассматриваются лишь поверхностно, но нужно отдавать себе отчет в их необходимости.

## 6.2.1. Задача о собирании купонов

В следующем разминочном упражнении по хешированию продолжим бросать мячики в эти $n$ контейнеров, пока не останется ни одного пустого контейнера, т. е. когда каждый контейнер не будет содержать, по крайней мере, один мячик. Сколько для этого следует ожидать бросков? Как можно видеть на рис. 6.9, для заполнения всех контейнеров может потребоваться значительно больше, чем $n$ бросков.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | 18 | |
| | | | | 32 | | | | 16 | |
| | 24 | | 29 | 31 | 19 | | | 15 | |
| | 21 | | 12 | 25 | 9 | | | 11 | 27 |
| 30 | 20 | 14 | 10 | 23 | 3 | | 26 | 4 | 22 |
| 28 | 13 | 8 | 5 | 17 | 1 | (33) | 7 | 2 | 6 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**Рис. 6.9.** Результаты эксперимента по бросанию мячиков в контейнеры случайным образом. Все контейнеры заполняются только к 33-му броску

Такую последовательность бросков можно разбить на $n$ прогонов, где каждый $r_i$ прогон состоит из бросков, выполненных после заполнения $i$ корзин и до следующего попадания в пустую корзину. Ожидаемое количество мячиков, требуемых для заполнения всех $n$ слотов $E(n)$, будет равным сумме ожидаемой длительности всех прогонов. При бросании монеты с вероятностью выпадения «орла», равной $p$, ожидаемое количество бросков до получения первого «орла» будет равным $1/p$. Это свойство *геометрического* распределения. После заполнения $i$ корзин вероятность попадания следующего броска в пустую корзину равна $p = (n - i)/n$. Собрав все это вместе, получим следующую формулу:

$$E(n) = \sum_{i=0}^{n-1} |r_i| = \sum_{i=0}^{n-1} \frac{n}{n-i} = n \sum_{i=0}^{n-1} \frac{1}{n-i} = nH_n \approx n \lg n.$$

Здесь главное помнить, что число гармоники

$$H_n = \sum_{i=1}^{n} 1/i$$

и что $H_n \approx \ln n$.

### Остановка для размышлений: Время покрытия для $K_n$

**ЗАДАЧА.** Предположим, что мы начинаем случайный обход полного $n$-вершинного графа (см. рис. 6.7, *б*) с вершины 1. На каждом шаге обхода мы направляемся с текущей вершины к произвольно выбранной соседней вершине. Каким будет ожидаемое количество шагов для посещения всех вершин графа?

**РЕШЕНИЕ**. Это точно такая же задача, как в предыдущем *разд. «Остановка для размышлений»*, но с графом другого типа, и поэтому, возможно, что ее решение будет иным.

В самом деле, наш случайный процесс генерирования произвольных целых чисел от 1 до $n$ выглядит по сути таким же, как и в задаче о собирании купонов. Это предполагает, что ожидаемая длительность покрывающего обхода будет $\Theta(n \log n)$. Единственная неувязка в этом доводе — модель случайного обхода не позволяет задерживаться в вершине на два последовательных шага, за исключением случаев, когда граф содержит ребра, выходящие и входящие в те же самые вершины (петли). Время покрытия графа без петель должно быть немного быстрее, поскольку повторное посещение вершин никоим образом не способствует обходу, но не настолько быстрее, чтобы изменить асимптотическую производительность. Вероятность нахождения одной из $(n - i)$ непосещенных вершин на следующем шаге меняется с $(n - i)/n$ на $(n - i)/(n - 1)$, уменьшая общее время покрытия с $nH_n$ до $(n - 1)H_n$. Но эти значения асимптотически одинаковые. Покрытие полного графа занимает $\Theta(n \log n)$ шагов, что намного быстрее, чем время покрытия маршрута. ■

## 6.3. Почему хеширование является рандомизированным алгоритмом?

Вспомним, что функция хеширования $h(s)$ сопоставляет ключи s целым числам от 0 до $m - 1$, в идеале равномерно распределяя их по этому интервалу. Поскольку качественная функция хеширования распределяет ключи по заданному диапазону целых чисел подобно генератору равномерно распределенных случайных чисел, хеширование можно анализировать, рассматривая его значения как результаты бросков игральной кости с $m$ гранями.

Но лишь сам факт возможности анализа хеширования в терминах теории вероятностей не делает его рандомизированным алгоритмом. Как утверждалось до сих пор, хеширование является полностью детерминированным процессом, не затрагивающим никаких случайных чисел. И действительно, хеширование должно быть детерминированным процессом, т. к. для того или иного значения $x$ каждый вызов функции хеширования $h(x)$ должен всегда выдавать абсолютно одинаковый результат, иначе мы никогда не найдем это значение $x$ в таблице хеширования.

Одна из причин, по которой мы отдаем предпочтение рандомизированным алгоритмам, это то, что они устраняют входной экземпляр наихудшего случая. Иными словами, плохая производительность должна быть результатом чрезвычайного невезения, а не вызываться подсунутыми каким-то шутником данными, заставляющими алгоритм творить несуразные вещи. Но для любой функции хеширования $h$ можно, в принципе, легко создать пример наихудшего случая. Возьмем, например, произвольное множество $S$, содержащее $nm$ различных ключей, и выполним хеширование каждого $s \in S$. Поскольку множество значений этой функции содержит только $m$ элементов, это должно вызывать много коллизий. Так как каждая корзина в среднем содержит $nm/m = n$ элементов, то из принципа Дирихле следует, что должна быть корзина, содержащая, по крайней мере, $n$ элементов. Эти $n$ элементов в этой корзине сами по себе окажутся кошмаром для функции хеширования $h$.

Как же можно избавиться от такого входного наихудшего случая? Этого можно добиться, выбирая нашу произвольную *функцию хеширования* из большого множества возможных функций, т. к. плохой входной экземпляр можно создать, только точно зная, какая функция хеширования будет использоваться.

Каким же образом можно создать семейство случайных функций хеширования? Вспомним, что обычно $h(x) = f(x) \pmod m$, где функция $f(x)$ преобразовывает ключ в огромное значение, которое сводится к требуемому диапазону функцией *mod m*, вычисляющей остаток от деления. Требуемый диапазон обычно определяется ограничениями приложения и памяти, поэтому выбор случайного $m$ нежелателен. Но что если предварительно уменьшить диапазон, используя большое целое число $p$? Обратите внимание на то, что, в общем,

$$f(x) \pmod m \neq (f(x) \bmod p) \pmod m.$$

Например:

$$21347895537127 \pmod{17} = 8 \neq (21347895537127 \pmod{2342343}) \pmod{17} = 12.$$

Таким образом, для определения функции хеширования можно выбрать случайное целое число $p$:

$$h(x) = ((f(x) \bmod p) \bmod m).$$

Результат будет вполне нормальным при следующих условиях:

- $f(x)$ — большое относительно $p$;
- $p$ — большое относительно $m$;
- $m$ и $p$ — взаимно простые.

Эта возможность выбора случайной функции хеширования означает, что теперь хеширование можно использовать для предоставления реальных рандомизированных гарантий, избавившись, таким образом, от входа наихудшего случая. Это также позволяет создавать мощные алгоритмы с использованием нескольких функций хеширования — таких как фильтры Блума, которые рассматриваются в *разд. 6.4*.

## 6.4. Фильтры Блума

Рассмотрим задачу выявления дубликатов документов, с которой часто приходится сталкиваться поисковым системам типа Google. Эти системы стремятся создать индекс всех *уникальных* документов в Интернете. Многие разные веб-сайты часто содержат идентичные копии документов, включая (к сожалению) пиратские копии этой книги. При сканировании следующей ссылки системе Google нужно определить, является ли найденный документ новым, стоящим добавления в индекс.

В такой ситуации, наверное, наиболее естественным решением будет создать таблицу хеширования для документов. Если следующий документ хешируется в пустую корзину, тогда он должен быть новым. Но наличие коллизии не обязательно означает, что мы уже имели дело с этим документом. Конечно же, новый документ нужно явно сравнить со всеми другими документами в контейнере, чтобы выявить ложные коллизии между $a$ и $b$, когда $h(a) = s$ и $h(b) = s$, но $a \neq b$. Этот момент обсуждается в *разд. 3.7.2*.

Но в этой ситуации ложные коллизии не такая и большая трагедия — они всего лишь означают, что Google не проиндексирует найденный новый документ. Это может быть приемлемо при условии достаточно низкой вероятности такого события. Отсутствие необходимости явного разрешения коллизий дает большую выгоду, позволяя уменьшить размер таблицы хеширования. Уменьшим размер корзины с указателя-связки до одного бита (указывающего, занята ли корзина или нет) — на типичной машине занимаемое таблицей пространство памяти уменьшится в 64 раза. Освобожденную таким образом память можно использовать, чтобы увеличить размер таблицы хеширования, тем самым изначально уменьшая вероятность коллизий.

Теперь предположим, что мы создали такую битово-векторную таблицу хеширования емкостью в $n$ битов. При наличии различных установленных битов, соответствующих $m$ документам, вероятность хеширования нового документа в один из этих битов будет равна $p = m/n$. Таким образом, даже если таблица заполнена только на 5%, вероятность ложной коллизии будет равна $p = 0{,}05$, что намного больше, чем приемлемо.

Намного лучшее решение — использовать *фильтр Блума*, который также является просто битово-векторной таблицей хеширования. Но, в отличие от простой хеш-таблицы, когда каждому документу соответствует в таблице одна позиция, фильтр Блума хеширует каждый ключ $k$ раз, используя $k$ разных функций хеширования. При подаче на вход фильтра Блума документа $s$ в таблице хеширования устанавливаются (присваиванием значения 1) все биты, соответствующие $h_1(s)$, $h_2(s)$, ... $h_k(s)$, означая тем самым, что эта позиция занята. Чтобы проверить, присутствует ли текущий исследуемый документ в таблице, необходимо проверить, равны ли все эти $k$ битов единице. Чтобы документ был ложно определен, как уже находящийся в фильтре, нужно, чтобы все его $k$ битов были установлены в хешах предыдущих документов, как показано на рис. ЦВ-6.10.

Каковы шансы подобного ложного определения? В таком фильтре Блума значения хешей для $m$ документов займут самое большее $km$ битов, поэтому вероятность одной коллизии возрастает до $p_1 = km/n$, что в $k$ раз больше, чем при использовании только одного значения хеша. Но коллизия должна произойти для всех $k$ битов таблицы с битами проверяемого документа, а вероятность этого события составляет всего лишь $p_k = (p_1)^k = (km/n)^k$. Это своеобразное выражение, поскольку вероятность, возведенная в $k$-ю степень, быстро уменьшается с увеличением $k$, но в нашем случае с увеличением $k$ вероятность повышается. Чтобы определить значение $k$, минимизирующее $p^k$, можно было бы взять производную и обнулить ее.

На рис. ЦВ-6.11 показан график этой вероятности ошибки в зависимости от нагрузки ($m/n$) с отдельной кривой для каждого значения $k$ от 1 до 5.

Очевидно, что использование большого количества функций хеширования (большое значение $k$) значительно уменьшает частоту ложных коллизий по сравнению с обычной таблицей хеширования (представленной на рис. 6.11 синей кривой с $k = 1$) — по крайней мере, для небольших нагрузок. Но обратите внимание на то, что частота ошибок для большего значения $k$ быстро повышается с увеличением нагрузки, поэтому для любой нагрузки всегда существует определенная точка, после которой добавление дополнительных функций хеширования становится неэффективным.

Для 5-процентной нагрузки частота ошибок для простой таблицы хеширования ($k = 1$) будет в 51,2 раза больше, чем для фильтра Блума с $k = 5$ ($9{,}77 \times 10^{-4}$), хотя обе эти таблицы занимают абсолютно одинаковый объем памяти. Фильтр Блума — замечательная структура данных для построения индексов при условии, что вы можете мириться со случающимися время от времени ошибками.

## 6.5. Парадокс дня рождения и идеальное хеширование

Таблицы хеширования являются замечательными практическими структурами данных для стандартных словарных операций вставки, удаления и поиска. Но время поиска хеша, достигающее в наихудшем случае величины $\Theta(n)$, вызывает раздражение, независимо от того, насколько редкими могут быть такие случаи. Нельзя ли каким-либо образом гарантировать постоянное время поиска в наихудшем случае?

Для *статических* словарей это возможно при использовании метода *идеального хеширования*. В этом методе мы получаем все ключи в одной партии, после чего добавлять или удалять элементы нельзя. Таким образом можно один раз создать структуру данных, а затем многократно использовать ее для поиска и/или проверки на вхождение во множество. Это довольно распространенное применение, и зачем тогда тратиться на динамические структуры данных, если они не требуются?

Один из способов реализации этого подхода — попробовать использовать заданную функцию хеширования $h(x)$ на нашем множестве $S$ из $n$ ключей и надеяться, что она создаст таблицу хеширования без коллизий, т. е. $h(x) \neq h(y)$, для всех пар различных $(x, y) \in S$. Должно быть очевидным, что шансы везения повышаются с увеличением размера таблицы относительно $n$ — чем больше пустых слотов доступно для следующего ключа, тем больше вероятность найти такой слот.

Какого размера должна быть таблица хеширования $m$, чтобы можно было ожидать отсутствия коллизий для $n$ ключей? Предположим, что мы начнем с пустой таблицы и будем многократно вставлять в нее ключи. Для $(i + 1)$-й вставки вероятность попадания в один из $(m - i)$ все еще свободных слотов в таблице составляет $(m - i)/m$. Для идеального хеширования все $n$ вставок должны попасть в пустой слот, поэтому

$$P(\textit{без коллизий}) = \prod_{i=0}^{n-1}\left(\frac{m-i}{m}\right) = \frac{m!}{m^n((m-n)!)}.$$

Наблюдаемый при этом вычислении феномен называется *парадоксом дня рождения*. Суть этого парадокса следующая. Сколько должно быть людей в комнате, чтобы существовала вероятность, что, по крайней мере, у двоих из них одинаковый день рождения? В этом случае размер таблицы хеширования $m = 365$. Вероятность отсутствия коллизий уменьшается до 1/2 для $n = 23$, а для $n \geq 50$ составляет меньше 3 процентов, как показано на рис. 6.12. Иными словами, при наличии в комнате только 23 людей коллизия (т. е. что двое из них имеют одинаковые дни рождения) вполне вероятна. Выполнив асимптотическую оценку, увидим, что коллизии можно ожидать, когда $n = \Theta\left(\sqrt{m}\right)$ или, что равносильно, когда $m = \Theta(n^2)$.

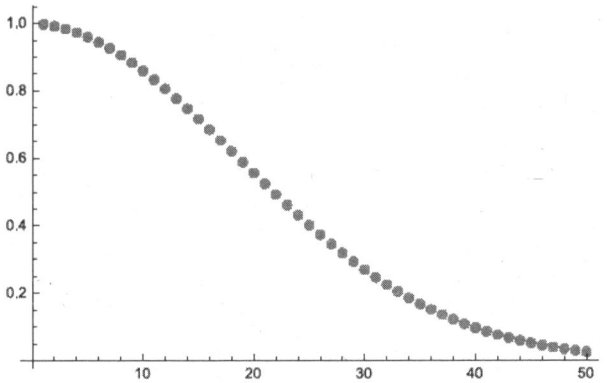



**Рис. 6.12.** Вероятность отсутствия коллизий быстро нарастает с увеличением количества ключей $n$ для хеширования. В рассматриваемом случае размер таблицы хеширования $m = 365$

Но квадратический объем памяти — слишком высокая цена за постоянное время доступа к $n$ элементам. Вместо этого создадим двухуровневую таблицу хеширования. Сначала $n$ ключей множества $S$ хешируются в таблицу с $n$ слотами. При этом коллизии будут вполне ожидаемыми, но, если только нам не сильно не повезет, все списки коллизий окажутся достаточно короткими.

Пусть $l_i$ обозначает длину $i$-го списка в этой таблице. Вследствие коллизий многие списки будут содержать более чем 1 элемент. Определим «достаточно короткий» список как $n$ элементов, распределенных по таблице таким образом, что сумма квадратов длин списков является линейной, т. е.

$$N = \sum_{i=1}^{n} l_i^2 = \Theta(n).$$

Предположим, так случилось, что все элементы находятся в списках длиной $l$, а это означает, что имеется $n/l$ непустых списков. Сумма квадратов длин этих списков будет: $N = (n/l)l^2 = nl$ — это линейная величина, поскольку $l$ является константой. Даже при наличии фиксированного количества списков длиной $\sqrt{n}$ можно все равно использовать линейный объем памяти.

Более того, можно доказать с высокой вероятностью, что $N \leq 4n$. Таким образом, если это свойство не выдерживается на $S$ для первой примененной функции, можно попробовать применить другую функцию. Весьма быстро обнаружится функция, дающая достаточно короткие длины списков, которую можно будет использовать.

Для таблицы второго уровня используем массив длиной $N$, выделив память объемом $l_i^2$ для элементов $i$-й корзины. Обратите внимание на то, что это — относительно количества элементов — достаточно большой объем, позволяющий избежать парадокса дня рождения: шансы в пользу отсутствия коллизий для любой используемой функции хеширования при этом повышаются. Если же коллизия и произойдет, просто пробует другую функцию хеширования, пока все элементы не будут распределены по отдельным слотам.

Полная схема этого подхода показана на рис. 6.13.

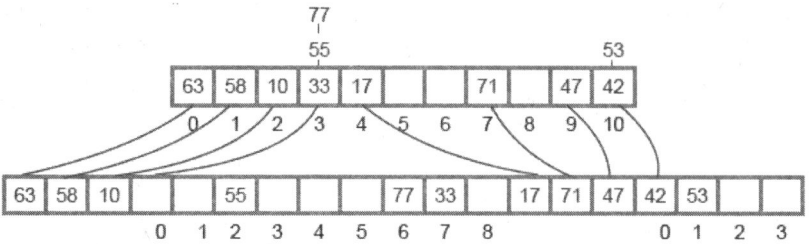



**Рис. 6.13.** В идеальном хешировании используется двухуровневая таблица хеширования, что гарантирует постоянное время поиска. В таблице первого уровня кодируется диапазон индексов элементов таблицы второго уровня количеством $l_i^2$, выделенных для хранения $l_i$ элементов в корзине $i$ первого уровня

Содержимое $i$-го элемента таблицы хеширования первого уровня содержит начальную и конечную позиции для $l_i^2$ элементов в таблице второго уровня, соответствующих этому списку. Здесь также находится описание (или идентификатор) функции хеширования, которая будет использоваться для этой области таблицы хеширования второго уровня.

Поиск элемента $s$ начинается с вызова функции хеширования $h_1(s)$ для вычисления правильного места в первой таблице, где будет находиться информация о соответствующей начальной и конечной позициях и параметры функции хеширования. Отсюда можно вычислить позицию $start + (h_2(s) \pmod{(stop - start)})$, в которой элемент $s$ будет находиться во второй таблице. Таким образом, поиск всегда можно выполнить за время $\Theta(1)$, используя линейный объем памяти двух таблиц.

Идеальное хеширование — очень полезная практическая структура данных, подходящая для всех случаев, когда выполняется большое количество запросов к статическому словарю. С базовой схемой можно выполнять большое количество настроек, чтобы минимизировать требуемый объем памяти и затраты на построение и поиск, — например, интенсифицировать работу по поиску функций хеширования второго уровня с меньшим количеством пропусков в таблице. Более того, минимальное идеальное хеширование гарантирует постоянное время доступа с нулевым количеством пустых слотов таблицы хеширования, в результате чего вторая таблица хеширования содержит $n$ элементов для $n$ ключей.

## 6.6. Метод минимальных хеш-кодов

Хеширование позволяет быстро проверить, находится ли конкретное слово $w$ из документа $D$ в документе $D_2$. Для этого просто хешируем все слова из документа $D_2$ в таблицу $T$, а затем в этой таблице ищем хеш $h(w)$ требуемого слова. Для простоты и эффективности из обоих документов можно удалить повторяющиеся слова, чтобы каждый из них содержал только по одному экземпляру каждой используемой словарной статьи. Проверив таким образом все статьи словаря $w_i \in D_1$ в таблице $T$, можно подсчитать количество элементов пересечения и вычислить *коэффициент сходства Жаккара* $J(D_1, D_2)$ этих двух документов:

$$J(D_1, D_2) = \frac{|D_1 \cap D_2|}{|D_1 \cup D_2|}.$$

Значение этого коэффициента может находиться в диапазоне от 0 до 1 — и представляет собой как бы вероятность того, что эти два документа совпадают.

Но что если нужно проверить сходство документов, не сравнивая все слова? Ведь при многократном сравнении, например в масштабах Интернета, эффективность играет важную роль. Предположим, что принять решение о степени схожести документов можно, только выбрав по одному слову из каждого документа. Какое слово следует выбрать в таком случае?

Первой мыслью может быть выбрать наиболее часто встречающееся слово в исходном документе, но, скорее всего, это будет слово «the», что даст очень мало информации о схожести документов. Будет лучше выбрать наиболее репрезентативное слово — например, по статистике TF-IDF (Term Frequency — Inverse Document Frequency — частота слова — обратная частота документа). Но и при этом подходе все равно делается предположение о распределении слов, что может быть неуместным.

Ключевая идея заключается в синхронизации, позволяющей выбрать одно и то же слово из каждого документа, исследуя их по отдельности. Эту задачу можно решить при помощи искусного, но простого приема, называемого методом *минимальных хеш-кодов* (minwise hashing). Суть этого приема следующая. Для каждого слова в документе $D_1$ вычисляется хеш-код $h(w_i)$, затем изо всех этих хеш-кодов выбирается код с наименьшим значением. Процесс повторяется для документа $D_2$, используя ту же функцию хеширования.

Этот подход привлекателен тем, что позволяет выбрать то же самое случайное слово в обоих документах, как показано на рис. 6.14.

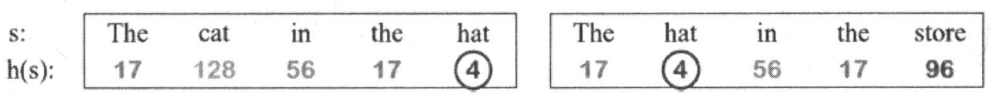



**Рис. 6.14.** Если два документа очень похожи, то слова, соответствующие хеш-кодам с минимальным значениями в каждом из этих документов, будут, скорее всего, одинаковыми

Предположим, что оба документа содержат идентичный словарный состав. Тогда слово с минимальным значением хеш-кода будет одинаковым в обоих документах $D_1$ и $D_2$, означая, что эти документы совпадают. В противоположность, предположим, что из каждого документа выбрано абсолютно случайное слово. Тогда вероятность выбора одинакового слова будет равна $1/v$, где $v$ обозначает размер словарного состава.

Теперь предположим, что словарные составы документов $D_1$ и $D_2$ не идентичны. Вероятность содержания слова с минимальным значением хеш-кода в обоих документах зависит от количества одинаковых слов, т. е. от пересечения их словарных составов, а также от общего размера словарного состава документов. По сути, эта вероятность — тот же коэффициент сходства Жаккара, описанный ранее.

Выборка большего количества слов — скажем, $k$ хеш-кодов с наименьшим значением из каждого документа, и возвращение размера пересечения по $k$ дает более точную оценку коэффициента сходства Жаккара. Но внимательный читатель может задаться вопросом: зачем утруждать себя? Вычисление всех хеш-кодов лишь для того, чтобы найти минимальные значения этих кодов, занимает линейное время относительно раз-

мера документов $D_1$ и $D_2$, тогда как вычисление точного размера пересечения при помощи хеширования заняло бы точно такое же время!

Ценность минимального значения хеш-кодов открывается при создании индексов для определения сходства и кластеризации по большим собраниям документов. Предположим, что имеется $N$ документов, каждый из которых содержит в среднем $m$ словарных статей. Мы хотим создать индекс, чтобы выбрать из этих документов наиболее сходный с документом $Q$.

Хеширование всех слов во всех документах дает таблицу размером $O(Nm)$. Чтобы сохранить же $k << m$ минимальных значений хеш-кодов для каждого документа, потребуется таблица намного меньшего размера — $O(Nk)$, но документы в пересечении корзин, связанных с $k$ минимальных хеш-кодов документа $Q$, будут, скорее всего, наиболее сходными, особенно при высоком коэффициенте сходства Жаккара.

**ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ:**

## Приблизительная оценка численности популяции

**ЗАДАЧА**. Предположим, что мы принимаем поток $S$ из $n$ чисел, по одному числу за раз. Поток содержит много дубликатов значений, возможно даже, что он состоит из одного числа, повторенного $n$ раз. Как можно оценочно установить количество различных значений в потоке $S$, используя только постоянный объем памяти?

**РЕШЕНИЕ**. Если бы объем памяти не был проблемой, естественным решением было бы создать словарную структуру данных для различных элементов потока, каждому из которых предоставляется количество его вхождений в поток. Если следующий считываемый из потока элемент уже присутствует в словаре, то инкрементируется его счетчик, а если нет, то он вставляется в словарь. Но согласно условиям задачи мы располагаем ограниченным объемом памяти для хранения постоянного количества элементов. Что мы можем делать в такой ситуации?

Выход состоит в использовании минимальных значений хеш-кодов. Предположим, что мы хешируем каждый входящий элемент $s$ потока $S$, но сохраняем хеш-код $h(s)$ только в том случае, если он меньше предыдущего минимального значения хеш-кода. Почему это представляет интерес? Предположим, что возможные значения хеш-кодов занимают диапазон от 0 до $M - 1$ и из этого диапазона мы равномерно выбираем $k$ случайных значений. Каким будет ожидаемое минимальное значение этих $k$ значений? Если $k = 1$, то ожидаемое значение будет (очевидно) $M/2$. На $k$, в общем, можно просто махнуть рукой и сказать, что если наши $k$ значений равномерно распределены по интервалу, то минимальное значение хеш-кода должно быть $M/(k + 1)$.

Такой подход — со взмахом рукой — тем не менее дает правильный ответ. Определим $X$, как наименьшее из $k$ выборок. Тогда

$$P(X = i) = P(X \geq i) - P(X \geq i+1) = \left(\frac{M-i}{M}\right)^k - \left(\frac{M-i-1}{M}\right)^k.$$

Взятие предела ожидаемого значения, когда $M$ стремится к большему значению, даст нам следующий результат:

$$E(X) = \sum_{i=0}^{M-1} i\left(\left(\frac{M-i}{M}\right)^k - \left(\frac{M-i-1}{M}\right)^k\right) \to \frac{M}{K+1}.$$

Суть в том, что значение $M$, разделенное на минимальное значение хеша, дает отличную прикидочную оценку количества различных увиденных нами значений. Этот метод нельзя сбить с толку повторяющимися значениями в потоке, поскольку функция хеширования будет создавать для всех таких повторов абсолютно одинаковые хеш-коды. ■

## 6.7. Эффективный поиск подстроки в строке

Строки представляют собой последовательности символов, в которых порядок размещения символов имеет значение, — так что строка АЛГОРИФМ (устаревший вариант слова «алгоритм») имеет совсем другое значение, чем строка ЛОГАРИФМ. Текстовые строки являются основным типом данных для многих компьютерных приложений: от синтаксического разбора и компилирования в языках программирования до поисковых систем Интернета и анализаторов биологических рядов.

Основной структурой данных для представления строк является массив символов, что обеспечивает постоянное время доступа к $i$-му символу строки. Для обозначения конца строки требуется хранить вместе с ней определенную вспомогательную информацию — специальный символ «конец строки» или (что, возможно, полезнее) количество символов $n$ в строке.

Одной из основных операций на текстовых строках является поиск подстроки в строке. В следующей задаче и ее решении приводится демонстрация такой операции.

**Задача**. Нахождение подстроки в строке

**Вход**. Текстовая строка $t$ и строка для поиска $p$.

**Выход**. Определить, содержит ли строка $t$ подстроку $p$, и если содержит, то в каком месте?

Самый простой алгоритм поиска подстроки в строке циклически накладывает подстроку $p$ на строку $t$, перемещаясь на одну позицию в строке $t$, пока последний символ подстроки $p$ не совместится с последним символом строки $t$, и проверяет на совпадение каждый символ подстроки $p$ с соответствующим символом строки $t$. Как было показано в *разд. 2.5.3*, время исполнения такого алгоритма равно $O(nm)$, где $n = |t|$ и $m = |p|$.

Это квадратичный предел времени исполнения в наихудшем случае. Но существуют алгоритмы и с линейным временем исполнения в наихудшем случае, но более сложные. Такие алгоритмы подробно рассматриваются в *разд. 21.3*. Здесь же мы рассмотрим алгоритм поиска подстроки с *ожидаемым* линейным временем исполнения, называемый *алгоритмом Рабина — Карпа* и основанный на использовании хеширования. Допустим, что мы вычислим определенную хеш-функцию от строки-образца $p$ и от подстроки длиной $m$ символов, начинающейся в позиции $i$ текста $t$. Очевидно, что если эти две строки одинаковые, то и их хеш-значения должны быть одинаковыми. Если же строки разные, то их хеш-значения *почти наверняка* будут разными. Одинаковые хеш-значения для разных строк должны встречаться настолько редко, что мы вполне можем позволить себе время $O(m)$ на явную проверку идентичности строк при совпадении хеш-значений.

Это сводит сложность поиска подстроки к $n - m + 2$ вычислениям хеш-значений ($n - m + 1$ хеш-значений для окон-подстрок в тексте $t$ плюс одно хеш-значение для строки-образца $p$) плюс несколько операций проверки с временем исполнения $O(m)$, количество которых *должно быть* очень небольшим. Но проблема состоит в том, что вычисление хеш-функции для строки из $m$ символов занимает время $O(m)$, а $O(n)$ таких вычислений, по-видимому, опять дают нам общее время исполнения алгоритма, равное $O(mn)$, в качестве оценки сложности.

Давайте подробнее рассмотрим применение нашей ранее (в *разд. 3.7*) определенной хеш-функции к $m$ символам строки $S$, начиная с позиции $j$:

$$H(S,j) - \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char\left(s_{i+j}\right).$$

Что изменится, если сейчас мы попробуем вычислить значение $H(S, j + 1)$ — т. е. хеш-значение следующей подстроки длиной в $m$ символов?

Обратите внимание, что символы $(m - 1)$ одинаковы в обеих подстроках, хотя количество умножений на $\alpha$ различается у них на единицу. Выполнив некоторые алгебраические преобразования, видим, что

$$H(S,j+1) = \alpha\left(H(S,j) - \alpha^{m-1} char\left(s_j\right)\right) + char\left(s_{j+m}\right).$$

Иными словами, когда нам известно хеш-значение подстроки, начинающейся в позиции $j$, то мы можем узнать хеш-значение подстроки с позиции $j + 1$, выполнив две операции умножения, одну операцию сложения и одну операцию вычитания. Эти операции можно выполнить за постоянное время (значение $\alpha^{m-1}$ можно вычислить один раз, после чего использовать его для всех вычислений хеш-значений). Такой подход годится даже для вычисления $H(S,j) \bmod M$, где $M$ является достаточно большим простым числом, тем самым ограничив величину наших хеш-значений (самое большее $M$) даже для длинных строк-образцов.

Алгоритм Рабина — Карпа (рис. ЦВ-6.15) является хорошим примером рандомизированного алгоритма (когда значение $M$ выбирается каким-либо произвольным способом). Время исполнения алгоритма не обязательно будет равно $O(n + m)$, т. к. нам может не повезти, и мы будем регулярно получать конфликтные хеш-значения для ложных совпадений. Тем не менее шансы в нашу пользу весьма высокие. Если хеш-функция возвращает значения равномерно в диапазоне от 0 до $M - 1$, то вероятность фальшивого совпадения и конфликта хеш-значений должна быть $1/M$. Это вполне приемлемо: если $M \approx n$, то должна быть только одна ложная коллизия на каждую строку, а если $M \approx n^k$ при $k \geq 2$, то существует очень высокая вероятность, что мы никогда не получим ложных коллизий.

## 6.8. Тестирование чисел на простоту

Одна из первых задач программирования, которую приходится решать студентам, — это проверка, является ли целое число $n$ простым, т. е. делится ли оно только на 1 и само себя. Последовательность простых чисел начинается с чисел 2, 3, 5, 7, 11, 13, 17... и продолжается до бесконечности.

Обычно программа для проверки числа на простоту основана на алгоритме деления, использующем цикл со значением счетчика $i$, инкрементируемым от 2 до $n - 1$, в котором проверяется, является ли частное $n/i$ целым числом. Если да, то число $n$ делится на $i$ без остатка и должно быть составным. Если же число $n$ не делится без остатка ни на одно из значений $i$, то оно будет простым. Вообще-то, счетчик цикла можно инкрементировать только до значения $i = \lceil \sqrt{n} \rceil$, т. к. это наибольшее возможное значение наименьшего нетривиального делителя числа $n$.

Тем не менее такие операции деления весьма затратны по времени. Если допустить, что каждая операция деления выполняется за постоянное время, то временна́я сложность алгоритма будет $O(\sqrt{n})$, но в нашем случае $n$ — это *значение* целого числа, над которым выполняется операция деления. Двоичное число с разрядностью, равной 1024 (размер небольшого ключа шифрования RSA), позволяет представлять числа величиной до $2^{1024} - 1$. Надежность шифрования RSA обеспечивается трудностью разложения числа ключа на множители. Обратите внимание, что $\sqrt{2^{1024}} = 2^{512}$, а это значение большее, чем количество атомов во вселенной. Поэтому будьте готовы подождать некоторое время, чтобы получить свой ответ.

Рандомизированные алгоритмы для тестирования простоты чисел (не полагающиеся на разложение на множители) оказываются намного быстрее. Согласно малой теореме Ферма, если $n$ — простое число, тогда

$$a^{n-1} = 1 (\text{mod } n)$$

для всех значений $a$, не делящихся на $n$.

Например, для $n = 17$ и $a = 3$: $(3^{17-1} - 1)/17 = 2\,532\,160$, поэтому $3^{17-1} = 1 (\text{mod } 17)$. Но для $n = 16$: $3^{16-1} = 11 (\text{mod } 16)$, что доказывает, что 16 не может быть простым числом.

Здесь интересно, что если $n$ является простым числом, то результат операции mod над этой большой степенью *всегда* будет равен 1. Это довольно ловкий прием, поскольку вероятность того, что этот результат будет равен 1, должна быть очень низкой — всего лишь $1/n$, *если* остаток распределен равномерно по диапазону.

Предположим, можно утверждать, что вероятность того, что остаток от деления составного числа будет 1, составляет меньше 1/2. Это подсказывает следующий алгоритм: выбираем 100 случайных целых чисел $a_j$, каждое в диапазоне от 1 до $n - 1$. Проверяем, что число $n$ не делится ни на одно из них, а затем вычисляем $(a_j)^{n-1} (\text{mod } n)$. Если результат всех этих делений равен 1, тогда вероятность того, что $n$ не является простым числом, должна быть меньше $(1/2)^{100}$ или ничтожно низкой. Так как исполняется фиксированное количество проверок (100), то время исполнения всегда быстрое, что делает этот алгоритм рандомизированным алгоритмом типа Монте-Карло.

Но с этим анализом вероятностей есть одна небольшая проблема. Оказывается, очень небольшое количество целых чисел (приблизительно одно из 50 миллиардов вплоть до числа $10^{21}$), не являющихся простыми, удовлетворяют малой теореме Ферма. Так, *числа Кармайкла* — например, 561 или 1105 — обречены на постоянную неправильную классификацию в качестве простых чисел. Тем не менее этот рандомизированный алгоритм очень эффективно отличает возможные простые числа от составных целых чисел.

***Подведение итогов***

Алгоритмы Монте-Карло всегда быстрые, обычно выдают правильное решение, и большинство из них ошибаются только в одном направлении.

Временная сложность вычисления $a^{n-1} \pmod n$ может быть моментом, вызывающим озабоченность. В действительности, время исполнения может быть $O(\log n)$. Вспомним, что $a^{2m}$ можно вычислить как $(a^m)^2$ методом «разделяй и властвуй», т. е. количество требуемых нам операций умножения может быть всего лишь логарифмическим по размеру показателя степени. Кроме того, для этого не требуется работать с чрезмерно большими числами. Из свойств модульной арифметики следует:

$$(x \cdot y) \bmod n = ((x \bmod n) \cdot (y \bmod n)) \bmod n.$$

Таким образом, в процессе вычислений нам никогда не понадобится умножать числа, большие чем $n$.

## 6.9. История из жизни. Как я дал Кнуту свой средний инициал

Великий Дональд Кнут — основополагающая фигура в создании теории вычислительных систем как интеллектуально отдельной академической дисциплины. Первые три тома его серии «Искусство программирования» (четыре — в настоящее время), изданные между 1968 и 1973 годами, показали математическую красоту проектирования алгоритмов и до сих пор представляют собой занимательный и захватывающий материал для чтения. Эти книги настолько полезные и увлекательные, что без каких бы то ни было сомнений я могу дать вам свое благословение отложить эту книгу и начать читать одну из книг Кнута — по крайней мере, на некоторое время.

Дональд Кнут также соавтор учебника «Конкретная математика», который фокусируется на методах математического анализа алгоритмов и дискретной математики. Подобно другим его книгам, кроме домашних заданий, этот учебник также содержит открытые вопросы по разным исследованиям. Мое внимание привлек один из таких вопросов, касающийся средних биномиальных коэффициентов и спрашивающий, верно ли следующее утверждение:

$$\binom{2n}{n} = (-1)^n \pmod{2n+1}$$

тогда и только тогда, когда $(2n + 1)$ является простым числом.

Этот вопрос напоминает малую теорему Ферма, рассматриваемую в *разд. 6.8* ранее.

Такое равенство легко доказывается для случаев, когда $(2n + 1)$ является простым числом. Выполнив базовые операции модульной арифметики, получим следующее:

$$(2n)(2n-1)\ldots(n+1) = (-1)(-2)\ldots(-n) = (-1)^n \cdot n! \pmod{(2n+1)}.$$

Так как $ad = bd \pmod m$ подразумевает $a = b \pmod m$, если $d$ — взаимно простое с $m$ и $(2n)!/n!$ делится на $n!$, то на $n!$ можно разделить обе стороны, получив этот результат.

Но верна ли эта формула *только* в тех случаях, когда $(2n + 1)$ является простым числом, как утверждается в задаче? Мне показалось, что это не совсем так, поскольку при-

веденная логика рассуждения отступает от алгоритма рандомизированной проверки простоты. Если рассматривать остаток операции mod $2n + 1$ как случайное целое число, вероятность, что оно может быть $(-1)^n$, очень низкая, всего лишь $1/n$. То есть отсутствие контрпримера при небольшом количестве проверок не представляет собой очень уж впечатляющее доказательство, поскольку случайные контрпримеры должны быть нечастыми.

Поэтому я создал программу из 16 строк на языке Mathematica и оставил ее исполняться в течение выходных. Когда я возвратился, программа остановилась на $n = 2953$. Оказывается, значение:

$$\binom{5906}{2953} \approx 7{,}93285 \times 10^{1775}$$

сравнимо с 5906 по модулю 5907. Но поскольку $5907 = 3 \cdot 11 \cdot 179$, это доказывает, что $(2n + 1)$ не является простым числом, и утверждение опровергается.

С чувством благоговейного трепета я сообщил этот результат самому Кнуту, который сказал, что он упомянет мое имя в следующем издании своей книги. Будучи пресловутым педантом, он попросил дать ему мой средний инициал[2]. Я гордо ответил «S» и спросил, когда он вышлет мне мой чек. Дело в том, что Дональд Кнут предлагал чек на $2,56 любому, кто найдет ошибку в одной из его книг[3], и я хотел получить такой чек как сувенир. Но он зарубил эту идею, объяснив, что решение открытой задачи не считается исправлением ошибки. Я до сих пор сожалею, что не сказал ему, что мой средний инициал «T». Тогда я мог бы указать ему на ошибку, которую нужно было бы исправить в следующем издании.

## 6.10. Откуда берутся случайные числа?

Все хитроумные рандомизированные алгоритмы, рассматриваемые в этой главе, порождают вопрос: а где именно мы берем для них случайные числа? Что происходит при вызове генератора случайных чисел используемого вами языка программирования?

Для генерирования случайных событий мы привыкли использовать физические процессы — такие как бросание монеты или игральных костей или даже отслеживание радиоактивного распада при помощи счетчика Гейгера. Мы верим, что эти события непредсказуемы и поэтому демонстрируют настоящую случайность.

Но программный генератор случайных чисел так не работает. Скорее всего, в нем используется, по сути, хеш-функция, именуемая *линейным конгруэнтным генератором*. В частности, следующее $n$-е случайное число $R_n$ является простой функцией предыдущего случайного числа $R_{n-1}$:

$$R_n = (aR_{n-1} + c) \bmod m,$$

где $a$, $c$, $m$ и $R_0$ представляют собой большие и тщательно подобранные константы. То есть мы хешируем предыдущее случайное число ($R_{n-1}$), чтобы получить следующее.

---

[2] *Англ.* middle initial — заглавная буква среднего (второго) имени в англоязычных странах. — *Прим. пер.*

[3] Если бы я когда-либо сделал такое предложение для моих книг, я бы разорился.

Дотошный читатель может задаться вопросом, насколько действительно случайными являются такие числа. В общем-то, они полностью предсказуемые, поскольку наличие числа $R_{n-1}$ предоставляет достаточно информации для воссоздания числа $R_n$. Такая предсказуемость означает, что упорный злоумышленник мог бы в принципе создать входной экземпляр наихудшего случая для рандомизированного алгоритма, если бы он знал текущее состояние генератора случайных чисел.

Будет более точным называть линейные конгруэнтные генераторы *генераторами псевдослучайных чисел*. Создаваемые этими генераторами числа выглядят случайными, т. к. они обладают такими же свойствами, какие можно было бы ожидать от чисел из действительно случайного источника. Этого обычно достаточно для хорошей работы рандомизированных алгоритмов на практике. Но при таком подходе утрачивается философское понятие случайности, что иногда оборачивается нежелательными последствиями, обычно в приложениях шифрования, гарантии безопасности которых основаны на предположении наличия настоящей случайности.

Генерирование случайных чисел — захватывающая задача. В *разд. 16.7* мы рассмотрим более подробно правильные и неправильные способы ее решения.

# Замечания к главе

Более формальное и основательное рассмотрение рандомизированных алгоритмов приводится в книге [MU17], а также в более раннем тексте [MR95]. Метод использования минимальных значений хеш-кодов изобрел Андрей Бродер (Andrei Z. Broder) — см. [Bro97].

# 6.11. Упражнения

## Вероятность

1. *[5]* Имеется $n$ правильных монет, которые нужно бросать до тех пор, пока они все не выпадут «орлом», используя следующий процесс: бросьте все $n$ монет в случайном порядке на стол. Затем соберите все монеты, выпавшие «решкой», и снова бросьте их на стол. В каждом раунде повторяйте этот процесс для всех монет, выпавших «решкой», до тех пор, пока все монеты не окажутся выпавшими «орлом».

   a) Какое ожидаемое количество раундов займет этот процесс?

   b) Какое ожидаемое количество бросков монет будет выполнено в процессе?

2. *[5]* Допустим, что мы бросаем $n$ монет, каждая из которых имеет известную неправильность, вследствие чего вероятность того, что $i$-я монета выпадет «орлом», составляет $p_i$. Разработайте эффективный алгоритм для определения точного значения вероятности выпадения точно $k$ «орлов», если дано $p_1, ..., p_n \in [0, 1]$.

3. *[5] Инверсией* перестановки называется пара элементов, расположенных в неправильном порядке.

   a) Покажите, что в перестановке из $n$ элементов может быть самое большее $n(n-1)/2$ инверсий. Какая перестановка (или перестановки) может содержать точно $n(n-1)/2$ инверсий?

b) Допустим, что $P$ является перестановкой, а $P^r$ — обращение этой перестановки. Покажите, что $P$ и $P^r$ содержат в точности $n(n-1)/2$ инверсий.

c) На основе предыдущего результата докажите, что ожидаемое количество инверсий в произвольной перестановке равно $n(n-1)/4$.

4. *[8] Беспорядком* (derangement) называется такая перестановка $p$ множества $\{1, ..., n\}$, в которой ни один элемент находится на своем правильном месте, т. е. $p_i \neq i$ для всех $1 \leq i \leq n$. Какова вероятность, что случайная перестановка будет беспорядком?

## Хеширование

5. *[легкая]* Радиостанция не проигрывает никаких иных записей, кроме Битлов, выбирая следующую песню случайным образом (заменяя равномерно). За час проигрывается около десяти песен. Я слушал их в течение 90 минут, прежде чем услышал повторяющуюся песню. Дайте приблизительную оценку, сколько песен записали Битлы.

6. *[5]* Для строк $S$ и $T$ длиной $n$ и $m$ соответственно найдите самую короткую подстроку в $S$, содержащую все символы в $T$ за ожидаемое время $O(n + m)$.

7. *[8]* Разработайте и внедрите эффективную структуру данных для поддержки кэша LRU (Least Recently Used — использованный наиболее давно) из $n$ целочисленных элементов. При полном заполнении кэша LRU из него удаляется использованный наиболее давно элемент. Для этого поддерживаются следующие операции:

   - `get(k)` — возвращается значение, связанное с ключом $k$, если он в настоящее время находится в кэше. В противном случае возвращается –1;
   - `put(k,v)` — если ключ $k$ существует, ему присваивается значение $v$. В противном случае $v$ вставляется как ключ $k$. Если очередь уже содержит $n$ элементов, прежде чем вставлять $(k, v)$, удаляем использованный наиболее давно элемент.

   Обе операции должны выполняться за ожидаемое время $O(1)$.

## Рандомизированные алгоритмы

8. *[5]* Пара английских слов $(w_1, w_2)$ называется *ротодромом* (rotodrome), если при циклическом смещении букв слова $w_1$ образуется слово $w_2$. Например, пара слов (*windup*, *upwind*) представляет собой ротодром, поскольку при смещении слова *windup* вправо на две позиции образуется слово *upwind*.

   Разработайте эффективный алгоритм для нахождения всех ротодромных пар для $n$ слов длиной $k$, предоставив для него временной анализ наихудшего случая. Также разработайте алгоритм с более быстрым ожидаемым временем исполнения, используя хеширование.

9. *[5]* Для массива $w$ положительных целых чисел, где $w[i]$ представляет вес индекса $i$, предложите алгоритм, который случайным образом выбирает индекс соответственно его весу.

10. *[5]* Функция `rand7` равномерно генерирует случайные целые числа в диапазоне от 1 до 7. Создайте на ее основе функцию `rand10`, которая равномерно генерирует случайные целые числа в диапазоне от 1 до 10.
11. *[5]* Пусть $0 < \alpha < 1/2$ — некая константа, независимая от входного массива размером $n$. Какова вероятность того, что подпрограмма разбиения из алгоритма быстрой сортировки выберет случайный разделительный элемент таким образом, что размер каждого из полученных подмассивов будет составлять как минимум размер исходного массива, умноженный на $\alpha$?
12. *[8]* Докажите, что для любой заданной нагрузки $m/n$ вероятность ошибки фильтра Блума минимальна при количестве функций хеширования, равном $k = \exp(-1)/(m/n)$.

## LeetCode

1. **https://leetcode.com/problems/random-pick-with-blacklist/**
2. **https://leetcode.com/problems/implement-strstr/**
3. **https://leetcode.com/problems/random-point-in-non-overlapping-rectangles/**

## HackerRank

1. **https://www.hackerrank.com/challenges/ctci-ransom-note/**
2. **https://www.hackerrank.com/challenges/matchstick-experiment/**
3. **https://www.hackerrank.com/challenges/palindromes/**

## Задачи по программированию

Эти задачи программирования доступны на сайте **https://onlinejudge.org**:

1. «Carmichael Numbers», глава 10, задача 10006.
2. «Expressions», глава 6, задача 10157.
3. «Complete Tree Labeling» глава 6, задача 10247.

# ГЛАВА 7

# Обход графов

Графы являются одной из важнейших областей теории вычислительных систем. Это абстрактное понятие, посредством которого можно описывать разнообразные реальные явления: организацию транспортных систем, человеческих взаимоотношений, сети передачи данных и т. п. Возможность формального моделирования такого множества разных реальных структур позволяет программисту решать широкий круг прикладных задач.

Более конкретно, граф $G = (V, E)$ состоит из набора *вершин* $V$ и набора *ребер* $E$, соединяющих пары вершин. С помощью графов можно представить практически *любые* взаимоотношения: создать, например, модель сети дорог, представляя населенные пункты вершинами, а дороги между ними — соединяющими соответствующие вершины ребрами (рис. 7.1, *а*), моделировать электрические схемы (рис. 7.1, *б*), представляя соединения компонентов вершинами, а компоненты — ребрами (или, альтернативно, компоненты — вершинами, а их соединения — ребрами), и т. п.

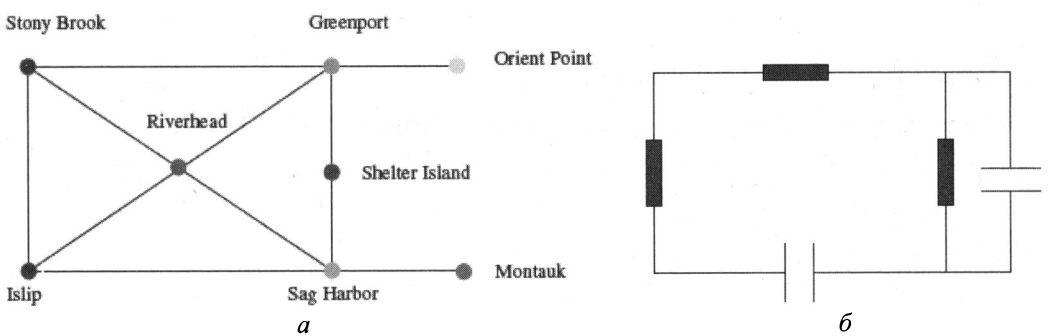



**Рис. 7.1.** Моделирование посредством графов сети дорог (*а*) и электрических схем (*б*)

Представление задачи в виде графа является ключевым подходом к решению многих алгоритмических задач. Теория графов предоставляет язык для описания свойств взаимоотношений, и просто поразительно, как часто запутанные прикладные задачи поддаются простому описанию и решению посредством применения свойств классических графов.

Разработка по-настоящему оригинальных алгоритмов на основе графов представляет собой трудную, но обычно необязательную задачу. Ключевым аспектом эффективного использования алгоритмов на графах является правильное моделирование задачи, чтобы можно было воспользоваться уже существующими алгоритмами. Знакомство с разными типами алгоритмических *задач* важнее знания деталей отдельных алгорит-

мов, особенно если учесть, что во второй части этой книги можно найти решение задачи по ее названию.

В этой главе рассматриваются основные структуры данных и операции обхода графов, которые можно использовать при поиске решений элементарных задач на графах. В *главе 8* речь пойдет о более сложных алгоритмах для работы с графами: построении минимальных остовных деревьев (minimum spanning tree), кратчайшего пути и потоков в сети. При этом правильное моделирование задачи остается наиболее важным аспектом ее решения. Затратив время на ознакомление с задачами и их решениями во второй части этой книги, вы сэкономите много времени при решении реальных задач в будущем.

# 7.1. Разновидности графов

Графом называется упорядоченная пара множеств $G = (V, E)$, где $V$ — подмножество вершин (или узлов), а $E$ — упорядоченное или неупорядоченное подмножество ребер, соединяющих пары вершин из $V$. Так, в моделировании дорожной сети вершины представляют населенные пункты (или пересечения дорог), определенные пары которых соединены дорогами (ребрами). В анализе исходного кода компьютерной программы вершины могут представлять операторы языка, а ребра — соединять операторы $x$ и $y$, если $y$ выполняется после $x$. В анализе человеческих взаимоотношений вершины представляют отдельных людей, а ребра соединяют тех, кто близок друг другу.

На выбор конкретного типа структуры данных для представления графов и алгоритмов для работы с ними оказывают влияние несколько фундаментальных свойств графов. Поэтому первым шагом в решении любой задачи на графах будет определение подходящего типа графа, с которым нужно будет работать. На рис. 7.2 показаны несколько основных типов графов, а далее дано их краткое описание.

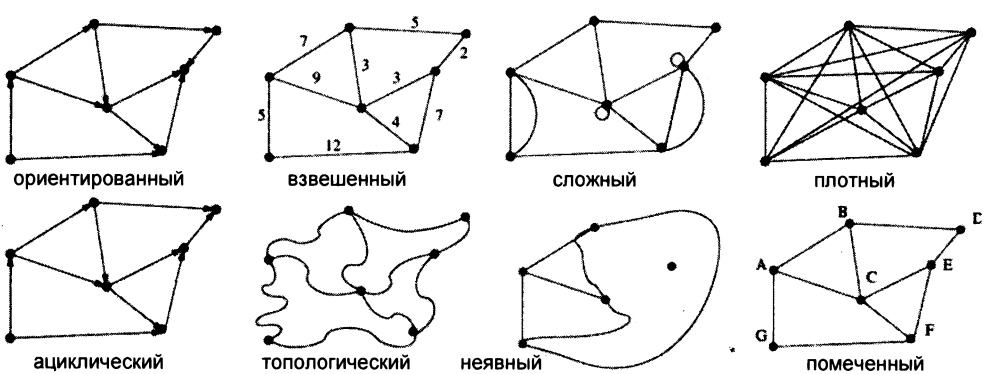



**Рис. 7.2.** Основные разновидности графов

- *Неориентированные/ориентированные.*

  Граф $G = (V, E)$ является *неориентированным* (undirected), если из обстоятельства, что ребро $(x, y) \in E$, следует, что ребро $(y, x)$ также является членом $E$. В противном случае говорят, что граф *ориентированный* (directed). Дорожные сети *между* горо-

дами обычно неориентированные, т. к. по любой обычной дороге можно двигаться в обоих направлениях. А вот дорожные сети *внутри* городов почти всегда ориентированы, поскольку в большинстве городов найдется, по крайней мере, несколько улиц с односторонним движением. Графы последовательности исполнения программ также обычно ориентированы, потому что программа исполняется от одной строчки кода к другой в одном направлении и меняет направление исполнения только при ветвлениях. Большинство графов, рассматриваемых в теории графов, являются неориентированными.

- *Взвешенные/невзвешенные.*

  Каждому ребру (или вершине) *взвешенного* (weighted) графа $G$ присваивается числовое значение, или вес. Например, в зависимости от приложения весом ребер графа дорожной сети может быть их протяженность, максимальная скорость движения, пропускная способность и т. п. Разные вершины и ребра *невзвешенных* (unweighted) графов не различаются по весу.

  Разница между взвешенными и невзвешенными графами становится особенно очевидной при поиске кратчайшего маршрута между двумя вершинами. В случае невзвешенных графов самый короткий маршрут состоит из наименьшего количества ребер, и его можно найти посредством *поиска в ширину*, рассматриваемого далее в этой главе. Поиск кратчайшего пути во взвешенных графах требует использования более сложных алгоритмов и описан в *главе 8*.

- *Простые/сложные.*

  Наличие ребер некоторых типов затрудняет работу с графами. *Петлей* (self-loop, loop) называется ребро $(x, x)$ — т. е. ребро, имеющее только одну вершину. *Кратными* (multiedge) называются ребра, соединяющие одну и ту же пару вершин $(x, y)$.

  Наличие в графе обеих этих структур (графы non-simple — *сложные*) требует особого внимания при реализации алгоритмов работы с ними, графы же, которые не содержат их, называются *простыми* (simple), или обыкновенными. Я сознаюсь, что все реализации в этой книге предназначены для работы **только** с простыми графами.

- *Разреженные/плотные.*

- Граф является *разреженным* (sparse), когда в действительности ребра определены только для малой части возможных пар вершин. Граф, у которого ребра определены для большей части возможных пар вершин, называется *плотным* (dense). Граф, содержащий все возможные ребра, называется *полным*. Для простого неориентированного графа количество всех возможных вершин составляет

$$\binom{n}{2} = (n^2 - n)/2.$$

  Не существует четкой границы между разреженными и плотными графами, но, как правило, плотные графы содержат $\Theta(n^2)$ ребер, а разреженные имеют линейный размер.

  Как правило, разреженность графов диктуется конкретным приложением. Графы дорожных сетей должны быть разреженными по причине физического и практического ограничения на количество дорог, которые могут пересекаться в одной точке.

На самом ужасном перекрестке, который мне когда-либо приходилось проезжать, сходилось всего лишь семь дорог. Подобным образом количество проводников электрической или электронной схемы, которые можно соединить в одной точке, также ограничено, за исключением, возможно, проводов питания или заземления.

- *Циклические/ациклические.*

  *Цикл* — это замкнутый контур из трех или более вершин, не содержащий повторяющихся вершин кроме начальной и конечной. *Ациклический* (acyclic) граф не содержит циклов. Связные ациклические неориентированные графы называются *деревьями*. Деревья представляют собой самые простые графы, рекурсивные по своей природе, поскольку разорвав любое ребро, мы получим два меньших дерева.

  Для обозначения ориентированных ациклических графов часто используется аббревиатура DAG (directed acyclic graph). Графы DAG обычно возникают в задачах календарного планирования, где ориентированное ребро $(x, y)$ обозначает, что мероприятие $x$ должно выполниться раньше, чем мероприятие $y$. Для соблюдения таких ограничений на предшествование вершины графа DAG упорядочиваются операцией, называющейся *топологической сортировкой* (topological sort). Топологическая сортировка обычно является первой выполняемой операцией в любом алгоритме на графе DAG (подробно об этом рассказано в *разд. 7.10.1*).

- *Вложенные/топологические.*

  Вершинно-реберное представление $G = (V, E)$ описывает чисто топологические аспекты графа. Граф называется *вложенным* (embedded), если его вершинам и ребрам присвоены геометрические позиции. Таким образом, любое визуальное изображение графа является *укладкой* (embedding), но это обстоятельство необязательно важно для алгоритма.

  Иногда структура графа полностью определяется геометрией его укладки. Например, если у нас имеется набор точек в плоскости и мы ищем самый короткий маршрут их обхода (решаем задачу коммивояжера), то в основе решения будет лежать *полный граф* (complete), т. е. граф, в котором каждая вершина соединена со всеми остальными. Веса обычно определяются расстоянием между каждой парой точек.

  Другим примером топологии графа, имеющей геометрическое происхождение, являются решетки точек. В большинстве задач по прямоугольной решетке размером $n \times m$ выполняется обход соседних точек, вследствие чего ребра определяются неявно, исходя из геометрии.

- *Неявные/явные.*

  Некоторые графы не создаются заранее с целью последующего обхода, а возникают по мере решения задачи. Хороший пример такого графа — граф поиска с возвратом. Вершины этого *неявного* (implicit) графа поиска представляют состояния вектора поиска, а ребра соединяют пары состояний, которые можно генерировать непосредственно друг из друга. Другой пример — анализ в масштабе Интернета, когда предпринимается попытка динамически посетить и проанализировать небольшую релевантную часть этой сети, вместо того, чтобы сразу загрузить все вебсайты. На рис. 7.2 показано это отличие уже известной части неявного графа от его остальной части, покрытой туманом, который рассеивается по мере исследования

этой части. Часто бывает, что работать с неявным графом проще, чем *явным* (explicit) образом создавать и сохранять весь граф для последующего анализа.

- *Помеченные/непомеченные.*

  В *помеченном* (labeled) графе каждая вершина имеет свою метку (идентификатор), что позволяет отличать ее от других вершин. В *непомеченных* (unlabeled) графах такие обозначения не применяются.

  Вершины графов, возникающие в прикладных задачах, часто помечаются значащими метками — например, названиями городов в графе транспортной сети. Распространенной задачей является проверка на изоморфизм, т. е. выяснение, является ли топологическая структура двух графов идентичной, принимая или не принимая во внимание метки (см. *разд. 19.9*).

## 7.1.1. Граф дружеских отношений

Чтобы продемонстрировать важность моделирования задачи должным образом, рассмотрим граф, в котором вершины представляют людей, а ребро, соединяющее две вершины, указывает, что между обозначаемыми этими вершинами людьми существуют дружеские отношения. Такие графы называются *социальными сетями*, и их можно четко определить для любого набора людей, будь то ваши соседи, однокурсники или коллеги, или жители всего земного шара. В последние годы возникла целая наука анализа социальных сетей, т. к. многие интересные аспекты поведения людей лучше всего поддаются пониманию на основе свойств графа дружеских отношений. Пример графа дружеских отношений между людьми представлен на рис. 7.3.

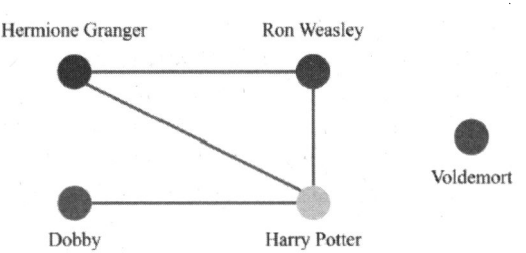



**Рис. 7.3.** Часть графа дружеских отношений персонажей книги о Гарри Поттере

Мы воспользуемся графами дружеских отношений, чтобы продемонстрировать описанную ранее терминологию графов, поскольку понимание терминологии является важной частью умения работать с графами.

- *Если я твой друг, то означает ли это, что ты мой друг?*

  Иными словами, мы хотим выяснить, является ли граф ориентированным. Граф называется *неориентированным*, если существование ребра $(x, y)$ всегда влечет за собой существование ребра $(y, x)$. В противном случае говорят, что граф является *ориентированным*. Граф типа «знаю о нем/о ней» является ориентированным, поскольку каждый из нас знает о многих людях, которые ничего не знают о нас. Граф типа «провел ночь с ним/ с ней» предположительно является неориентированным,

т. к. для такого занятия требуется партнер. Лично мне хотелось бы, чтобы граф дружеских отношений также был неориентированным.

- *Насколько ты хороший друг?*

  Во *взвешенных* графах каждому ребру присваивается числовой атрибут. Мы можем смоделировать уровень дружеских отношений, присвоив каждому ребру соответствующее числовое значение — например, от –100 (враги) до 100 (родные братья). Также, в зависимости от приложения, весом ребер графа дорожной сети может быть их протяженность, максимальная скорость движения, пропускная способность и т. п. Граф называется *невзвешенным*, если все его ребра имеют одинаковый вес.

- *Друг ли я сам себе?*

  Иными словами, мы хотим выяснить, является ли граф *простым*, т. е. не содержит петель и кратных ребер. *Петлей* называется ребро типа $(x, x)$. Иногда люди поддерживают несколько типов отношений друг с другом. Например, возможно, $x$ и $y$ были однокурсниками в институте, а сейчас работают вместе на одном предприятии. Такие взаимоотношения можно моделировать посредством кратных ребер, снабженных разными метками.

  Поскольку с простыми графами легче работать, то обычно нам лучше объявить, что никто не является другом самому себе.

- *У кого больше всех друзей?*

  *Степенью* вершины называется количество ее ребер. В графе дружеских отношений наиболее общительный человек определяется нахождением вершины наивысшей степени, а одинокие отшельники — вершинами нулевой степени.

  В *плотных* графах большинство вершин имеет высокую степень, в отличие от *разреженных* графов, в которых количество ребер сравнительно небольшое. В *регулярных*, или однородных, графах все вершины имеют одинаковую степень. Регулярный граф дружеских отношений представляет по-настоящему предельный уровень социальности.

  Для решения реальных задач в большинстве случаев применяются графы разреженного типа. Граф дружеских отношений является хорошим примером разреженных графов. Даже самый общительный человек в мире знает лишь незначительную часть населения планеты.

- *Живут ли мои друзья рядом со мной?*

  Социальные сети находятся под сильным влиянием географического фактора. Многие из наших друзей являются таковыми лишь потому, что они просто живут рядом с нами (например, соседи) или когда-то жили вместе с нами (например, старые соседи по комнате в студенческом общежитии).

  Таким образом, для полного понимания социальной сети требуется вложенный граф, где каждая вершина связана с географической точкой, в которой живет конкретный член сети. Эта географическая информация может быть и не закодирована в графе явным образом, но тот факт, что граф дружеских отношений по своей природе является *вложенным* на поверхности сферы, влияет на нашу интерпретацию сети.

- *О, ты ее тоже знаешь?*

  Службы социальных сетей, таких как Instagram или LinkedIn, *явно* определяют дружеские связи между членами этих сетей. Графы таких социальных сетей состоят из направленных ребер от члена (вершины) *x*, заявляющего о своей дружбе с другим членом, к этому члену (вершине) *y*.

  С другой стороны, действительный граф дружеских отношений всего населения Земли является *неявным* по той причине, что каждый знает, кто его друг, но не может знать о дружеских отношениях других людей. *Гипотеза шести шагов* утверждает, что любые два человека в мире (например, профессор Скиена и президент США) связаны короткой цепочкой из промежуточных людей, но не предоставляет никакой информации, как именно определить эту связывающую цепочку. Самый короткий известный мне путь содержит три звена: Стивен Скиена — Марк Фасиано — Майкл Ашнер — Дональд Трамп (Steven Skiena — Mark Fasciano — Michael Ashner — Donald Trump). Но может существовать и более короткий, неизвестный мне, путь — например, если Дональд Трамп учился в колледже вместе с моим дантистом[1]. Но из-за того, что граф дружеских отношений является неявным, проверить эту или другие возможные цепочки не так-то просто.

- *Вы действительно личность или всего лишь лицо в толпе?*

  Этот вопрос сводится к тому, является ли граф дружеских отношений *помеченным* или нет. То есть имеют ли важность для нашего анализа имена или ID-номера вершин?

  Во многих исследованиях социальных сетей метки графов не играют большой роли. В качестве метки вершинам графа часто присваивается порядковый номер, что обеспечивает удобство идентификации и в то же время сохраняет анонимность представляемого ею лица. Вы можете протестовать, что номер вас обезличивает и у вас есть имя, но попробуйте доказать это программисту, который реализует алгоритм. В графе дружеских отношений, отображающем распространение слухов или инфекционного заболевания, можно исследовать такие свойства сети, как связность, распределение степеней вершин графа или распределение длин путей. Эти свойства не претерпевают никаких изменений вследствие смешивания ID-номеров вершин.

### Подведение итогов

С помощью графов можно моделировать большое разнообразие структур и взаимоотношений. Для работы с графами и обмена информацией о них используется терминология теории графов.

[1] Впрочем, существует и путь: Стивен Скиена — Стив Исраэль — Джо Байден (Steven Skiena — Steve Israel — Joe Biden), так что я подстраховался на случай, если он выиграет выборы в 2020 г. (теперь мы знаем, что именно так и произошло).

## 7.2. Структуры данных для графов

Выбор правильной структуры данных для графа может иметь огромное влияние на производительность алгоритма. Двумя основными структурами данных для графов являются *матрицы смежности* (adjacency matrix) и *списки смежности* (adjacency list), показанные на рис. ЦВ-7.4.

Предположим, что граф $G = (V, E)$ содержит $n$ вершин и $m$ ребер.

- *Матрица смежности.*

  Граф $G$ можно представить с помощью матрицы $M$ размером $n \times n$, где элемент $M[i, j] = 1$, если $(i, j)$ является ребром графа $G$, и 0 в противном случае. Таким образом мы можем дать быстрый ответ на вопрос: «Содержит ли граф $G$ ребро $(i, j)$?», а также быстро отобразить вставки и удаления ребер. Но такая матрица может потребовать большого объема памяти для графов с большим количеством вершин и относительно небольшим количеством ребер.

  Рассмотрим, например, граф, представляющий карту улиц Манхэттена. Каждое пересечение улиц отображается на графе в виде вершины, а соседние пересечения соединяются ребрами. Каким будет размер такого графа? Уличная сеть Манхэттена, по сути, представляет собой решетку из 15 авеню, пересекаемых 200 улицами. Это дает нам около 3000 вершин и 6000 ребер, поскольку почти все вершины соседствуют с четырьмя другими вершинами, а каждое ребро является общим для двух вершин. Эффективное сохранение такого скромного объема данных не должно вызывать никаких проблем, но матрица смежности заняла бы $3000 \times 3000 = 9\,000\,000$ элементов, почти все из которых были бы пустыми.

  Можно было бы сэкономить некоторый объем памяти, упаковывая в одном слове несколько битов состояния или используя структуру данных симметричной матрицы (например, треугольную матрицу) для неориентированных графов. Но с использованием этих методов теряется простота, которая делает матрицу смежности такой привлекательной, и, что более критично, время исполнения даже для разреженных графов остается по своей сути квадратичным.

- *Списки смежности.*

  Более эффективным способом представления разреженных графов является использование связных списков для хранения соседствующих вершин. Хотя списки смежности требуют использования указателей, вы легко с ними справитесь, когда наберетесь немного опыта работы со связными структурами данных.

  В списках смежности проверку на присутствие определенного ребра $(i, j)$ в графе $G$ выполнить труднее, поскольку, чтобы найти это ребро, необходимо выполнить поиск по соответствующему списку. Однако разработать алгоритм, не нуждающийся в таких запросах, на удивление легко. Обычно перебираются все ребра графа при одном его обходе в ширину или глубину, и при посещении текущего ребра обновляется его состояние. Преимущества той или иной структуры смежности для представления графов приведены в табл. 7.1.

**Таблица 7.1.** Сравнение матриц и списков смежности

| Задача | Оптимальный вариант |
|---|---|
| Проверка на вхождение ребра $(x, y)$ в граф | Матрица смежности |
| Определение степени вершины | Списки смежности |
| Объем памяти для разреженных графов | Списки смежности $(m - n)$ против матрицы смежности $(n^2)$ |
| Объем памяти для плотных графов | Матрица смежности (с небольшим преимуществом) |
| Вставка или удаление ребра | Матрица смежности $O(1)$ против списков смежности $O(d)$ |
| Обход графа | Списки смежности $\Theta(m + n)$ против матрицы смежности $\Theta(n^2)$ |
| Пригодность для решения большинства проблем | Списки смежности |

### *Подведение итогов*

Для большинства приложений на графах списки смежности являются более подходящей структурой данных, чем матрицы смежности.

В этой главе для представления графов мы будем использовать списки смежности. В частности, представление графа осуществляется таким образом: для каждого графа ведется подсчет количества вершин, каждой из которых присваивается идентификационный номер в диапазоне от 1 до $n$. Ребра представляются посредством массива связных списков. Соответствующий код приведен в листинге 7.1.

**Листинг 7.1. Реализация графов посредством списков смежности**

```
#define MAXV 100            /* Максимальное количество вершин */

typedef struct edgenode {
    int y;                  /* Информация о смежности */
    int weight;             /* Вес ребра, если есть */
    struct edgenode *next;  /* Следующее ребро в списке */
} edgenode;

typedef struct {
    edgenode *edges[MAXV+1];  /* Информация о смежности */
    int degree[MAXV+1];       /* Степень каждой вершины */
    int nvertices;            /* Количество вершин в графе */
    int nedges;               /* Количество ребер в графе */
    int directed;             /* Граф ориентированный? */
} graph;
```

Ориентированное ребро $(x, y)$ представляется в списке смежности структурой типа `edgenode`. Поле `degree` содержит степень соответствующей вершины. Неориентирован-

ное ребро $(x, y)$ входит дважды в любою структуру графа на основе смежности: один раз в виде $y$ в списке для $x$ и второй раз в виде $x$ в списке для $y$. Булев флаг `directed` указывает, является ли данный граф ориентированным.

Для демонстрации использования этой структуры данных мы покажем, как выполнить считывание графа из файла. Типичный формат файла графа таков: первая строчка содержит количество вершин и ребер в графе, а за ней следуют строчки, в каждой из которых указаны вершины очередного ребра. Начинаем с инициализации структуры (листинг 7.2).

**Листинг 7.2. Формат графа**

```
void initialize_graph(graph *g, bool directed) {
    int i;                    /* Счетчик */

    g->nvertices = 0;
    g->nedges = 0;
    g->directed = directed;

    for (i = 1; i <= MAXV; i++) {
        g->degree[i] = 0;
    }

    for (i = 1; i <= MAXV; i++) {
        g->edges[i] = NULL;
    }
}
```

Собственно чтение файла графа заключается во вставке каждого ребра в эту структуру (листинг 7.3).

**Листинг 7.3. Считывание графа**

```
void read_graph(graph *g, bool directed) {
    int i;                    /* Счетчик */
    int m;                    /* Количество вершин */
    int x, y;                 /* Вершины в ребре (x,y) */

    initialize_graph(g, directed);

    scanf("%d %d", &(g->nvertices), &m);

    for (i = 1; i <= m; i++) {
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, directed);
    }
}
```

В листинге 7.3 критичной является процедура `insert_edge`. Новый узел `edgenode` вставляется в начало соответствующего списка смежности, т. к. порядок не имеет значения.

Процедуре вставки передается параметр в виде булева флага `directed` для обозначения, сколько копий каждого ребра нужно вставить: одну или две. Код процедуры вставки приведен в листинге 7.4. Обратите внимание на использование рекурсии для вставки копии.

**Листинг 7.4. Вставка ребра**

```
void insert_edge(graph *g, int x, int y, bool directed) {
    edgenode *p;                    /* Временный указатель */

    p = malloc(sizeof(edgenode)); /* Выделяем память для edgenode */

    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    g->edges[x] = p;                /* Вставка в начало списка */

    g->degree[x]++;

    if (!directed) {
        insert_edge(g, y, x, true);
    } else {
        g->nedges++;
    }
}
```

Для вывода графа на экран достаточно двух вложенных циклов: одного — для вершин, другого — для их смежных ребер (листинг 7.5).

**Листинг 7.5. Вывод графа на экран**

```
void print_graph(graph *g) {
    int i;                  /* Счетчик */
    edgenode *p;            /* Временный указатель */

    for (i = 1; i <= g->nvertices; i++) {
        printf("%d: ", i);
        p = g->edges[i];
        while (p != NULL) {
            printf(" %d", p->y);
            p = p->next;
        }
        printf("\n");
    }
}
```

Было бы разумно использовать хорошо спроектированный тип данных графа в качестве модели для создания своего собственного, а еще лучше в качестве основы для разрабатываемого приложения. Я рекомендую обратить внимание на библиотеки типов

LEDA (см. *разд. 22.1.1*) или Boost (см. *разд. 22.1.3*), которые я считаю лучше всего спроектированными структурами данных графов общего назначения, имеющимися в настоящее время. Эти библиотеки наверняка окажутся более мощными (и, следовательно, будут иметь больший размер и замедлять работу в большей степени), чем вам нужно, но они предоставляют такое количество правильно работающих функций, которые вы, вероятнее всего, не сможете реализовать самостоятельно с такой степенью эффективности.

## 7.3. История из жизни. Жертва закона Мура

Я являюсь автором Combinatorica (**www.combinatorica.com**) — библиотеки алгоритмов для работы с графами (рис. 7.5), предназначенной для взаимодействия с компьютерной системой алгебраических вычислений Mathematica. Большой проблемой в Mathematica является производительность — из-за применяемой в ней аппликативной модели вычислений (в ней не поддерживаются операции записи в массивы с постоянным временем исполнения) и накладных расходов на интерпретацию кода (в отличие от компилирования). Код на языке программирования пакета Mathematica обычно исполняется от 1000 до 5000 раз медленнее, чем код на языке C.

Эти особенности приложения могут значительным образом замедлять его работу. Что еще хуже, Mathematica сама занимает большой объем оперативной памяти, требуя для эффективной работы целых четыре мегабайта, что в 1990 году, когда я завершил работу над Combinatorica, было потрясающе много. При этом любая попытка вычислений на достаточно больших структурах неизбежно вызывала переполнение памяти. В такой среде мой графический пакет имел надежду эффективно работать только на графах *очень* небольшого размера.

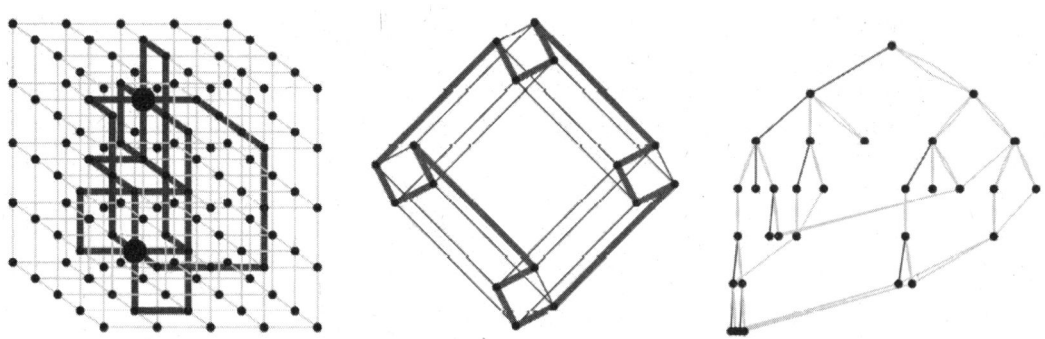


**Рис. 7.5.** Репрезентативные графы в Combinatorica: пути, не имеющие общих ребер (*слева*), гамильтонов цикл в гиперкубе (*в центре*), обход в глубину дерева поиска (*справа*)

Одним из проектных решений, принятых мною вследствие описанных недостатков Mathematica, было использование матриц смежности вместо списков смежности в качестве основной структуры данных графов для Combinatorica. Такое решение может сначала показаться странным. Разве в ситуации нехватки памяти не выгоднее использовать списки смежности, экономя каждый байт? Вообще говоря, да, но ответ не будет

таким простым в случае очень малых графов. Представление взвешенного графа, состоящего из $n$ вершин и $m$ ребер, посредством списка смежности требует приблизительно $n + 2m$ слов, где составляющая $2m$ отражает требования к памяти для хранения информации о конечных точках и весе каждого ребра. Таким образом, использование списков смежности предоставляет преимущество в отношении объема требуемой памяти только в том случае, если выражение $n + 2m$ значительно меньше, чем $n^2$. Размер матрицы смежности остается управляемым для $n \leq 100$, и, конечно же, для плотных графов этот размер вдвое меньше, чем размер списков смежности.

Для меня более насущной заботой была необходимость минимизации накладных расходов, обусловленных использованием интерпретируемого языка. Результаты эталонных тестов представлены в табл. 7.2.

**Таблица 7.2.** Результаты эталонных тестов старой Combinatorica на рабочих станциях пяти поколений (время исполнения в секундах)

| Команда | Машина | | | | |
|---|---|---|---|---|---|
| | Sun-3 | Sun-4 | Sun-5 | Ultra 5 | Sun Blade |
| `PlanarQ[GridGraph[4,4]]` | 234,10 | 69,65 | 27,50 | 3,60 | 0,40 |
| `Length [Partitions[30]]` | 289,85 | 73,20 | 24,40 | 3,44 | 1,58 |
| `VertexConnectivity [GridGraph[3,3]]` | 239,67 | 47,76 | 14,70 | 2,00 | 0,91 |
| `RandomPartition[1000]` | 831,68 | 267,5 | 22,05 | 3,12 | 0,87 |

В 1990 году решение двух довольно сложных, но имеющих полиномиальное время выполнения задач на графах из 9 и 16 вершин занимало на моем настольном компьютере несколько минут! Квадратичный размер структуры данных определенно не мог оказать большого влияния на время исполнения этих задач, т. к. $9 \times 9$ равно всего лишь 81. По своему опыту я знал, что язык программирования пакета Mathematica работает лучше со структурами данных постоянного размера, наподобие матриц смежности, чем со структурами данных, имеющими непостоянный размер, какими являются списки смежности.

Тем не менее, несмотря на все эти проблемы с производительностью, библиотека Combinatorica оказалась очень хорошим приложением, и тысячи людей использовали этот пакет для всевозможных интересных экспериментов с графами. Combinatorica никогда не претендовала на звание высокопроизводительной библиотеки алгоритмов. Большинство пользователей быстро осознали, что вычисления на больших графах нереальны, но тем не менее с энтузиазмом работали с этой библиотекой как с инструментом для математических исследований и среды моделирования. Все были довольны.

Но по прошествии нескольких лет пользователи Combinatorica начали спрашивать, почему вычисления на графах небольшого размера занимают так много времени. Это меня не удивляло, т. к. я знал, что моя программа всегда была медленной. Но почему людям потребовались годы, чтобы заметить это?

Объяснение заключается в том, что скорость работы компьютеров удваивается приблизительно каждые два года. Это явление носит названия *закона Мура*. *Ожидания* пользователей относительно производительности прикладных программ возрастают в соответствии с этими улучшениями в аппаратном обеспечении. Частично из-за того, что Combinatorica была рассчитана на работу со структурами данных графов квадратичного размера, она недостаточно хорошо масштабировалась на разреженные графы.

С годами требования пользователей становились все жестче, и, наконец, я решил, что Combinatorica нуждается в обновлении. Мой коллега, Срирам Пемараджу (Sriram Pemmaraju), предложил мне свою помощь. Спустя десять лет после первоначального выпуска библиотеки мы (преимущественно он) полностью переписали Combinatorica, воспользовавшись более быстрыми структурами данных.

В новой версии Combinatorica для хранения графов используется очень эффективная структура данных в виде списка ребер. Размер списков ребер, как и размер списков смежности, линейно зависит от размера графа (ребра плюс вершины). Это заметно улучшило производительность большинства функций для работы с графами. Повышение производительности особенно драматично для «быстрых» алгоритмов обработки графов. Это алгоритмы с линейным или почти линейным временем исполнения: алгоритмы обхода графов, топологической сортировки и поиска компонентов связности или двусвязности. Последствия этой модификации проявляются во всем пакете в виде уменьшения времени работы и более экономного расхода памяти. Теперь Combinatorica может обрабатывать графы, которые в 50–100 раз больше, чем те, с которыми могла работать старая версия.

Для сравнения производительности новой и старой версий использовались разреженные (решетчатые) графы, разработанные специально, чтобы выделить разницу между этими двумя структурами данных. Да, новая версия библиотеки *намного* быстрее, но обратите внимание, что разница в производительности становится заметной только для графов большего размера, чем те, для работы с которыми предназначалась старая версия Combinatorica. Так, на рис. 7.6, *а* приводится график времени исполнения функции `MinimumSpanningTree` для обеих версий Combinatorica. Но относительная разница во времени исполнения увеличивается с возрастанием $n$. На рис. 7.6, *б* показано соотношение времени исполнения старой и новой версии в зависимости от размера графа. Разница между линейным и квадратичным ростом размера является асимптотической, поэтому с возрастанием $n$ последствия перехода на новую версию становятся еще более заметными.

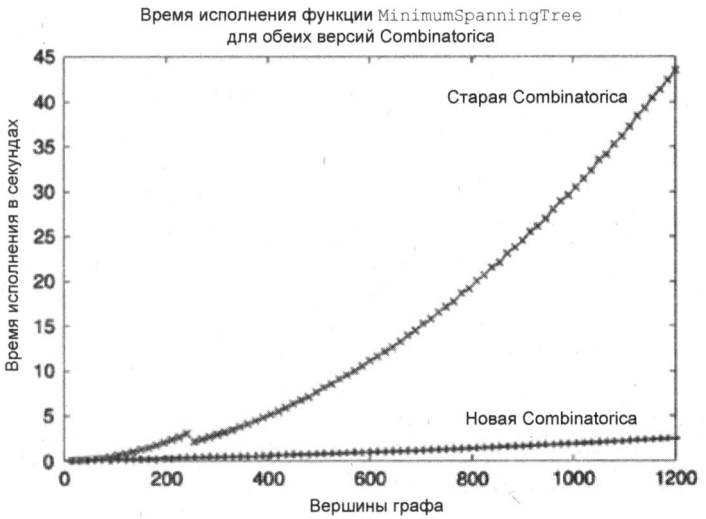
Обратите внимание на странный всплеск на графике при $n \approx 250$. Скорее всего, это следствие перехода между разными уровнями иерархии памяти. Такие явления не редкость в современных сложных компьютерных системах. При разработке структур данных производительность кэша должна быть одним из главных, но не важнейшим из принимаемых во внимание обстоятельств.

Повышение производительности, достигнутое благодаря использованию списков смежности, намного превышает любое улучшение, обусловленное применением кэша.

Из нашего опыта разработки и модифицирования библиотеки Combinatorica можно извлечь такие три основных урока.

- *Чтобы ускорить время исполнения программы, нужно лишь подождать некоторое время.*

Передовое аппаратное обеспечение со временем доходит до пользователей всех уровней. В результате улучшений в аппаратном обеспечении, имевших место в течение 15 лет, скорость работы первоначальной версии библиотеки Combinatorica возросла более чем в 200 раз. В этом контексте дальнейшее повышение производительности вследствие модернизации библиотеки является особенно значительным.




*а*




*б*

**Рис. 7.6.** Сравнение производительности старой и новой версий Combinatorica: абсолютное время исполнения для каждой версии (*а*) и соотношение времени исполнения (*б*)

- *Асимптотика в конечном счете является важной.*

  Я не сумел предвидеть развитие технологии, и это было ошибкой с моей стороны. Хотя никто не может предсказывать будущее, можно с довольно большой степенью уверенности утверждать, что в будущем компьютеры станут обладать большим объемом памяти и работать быстрее, чем компьютеры сейчас. Это дает преимущество более эффективным алгоритмам и структурам данных, даже если сегодня их производительность не намного выше, чем у альтернативных решений. Если сложность реализации вас не останавливает, подумайте о будущем и выберите самый лучший алгоритм.

- *Постоянные факторы могут быть важными.*

  В связи с возрастающей важностью изучения сетей организация Wolfram Research недавно переместила базовые структуры данных графов в ядро приложения Mathematica. Это позволяет записывать их на компилированном, а не интерпретированном языке, что ускоряет все операции приблизительно в 10 раз по сравнению с библиотекой Combinatorica.

  Ускорение вычислений в десять раз часто имеет большую важность, означая один день вычислений вместо одной недели или один месяц вместо одного года. В этой книге основное внимание уделяется асимптотической сложности вычислений, т. к. ее цель состоит в обучении фундаментальным принципам. Но на практике константы также могут иметь важность.

## 7.4. История из жизни. Создание графа

— Только на чтение данных у этого алгоритма уходит пять минут. На получение сколько-нибудь интересных результатов не хватит *никакого* времени.

Молодая аспирантка была полна энтузиазма, но не имела ни малейшего понятия о мощи правильно выбранных структур данных.

Как было описано в *разд. 3.6*, мы экспериментировали с алгоритмами для разбиения сетки треугольников на полосы для быстрого рендеринга триангулированных поверхностей. Задачу поиска наименьшего количества полос, которые покрывают каждый треугольник в сетке, можно смоделировать как задачу на графах. В таком графе каждый треугольник сетки представляется вершиной, а смежные треугольники — ребрами между соответствующими вершинами. Такое представление в виде *двойственного графа* содержит всю информацию, необходимую для разбиения треугольной сетки на полосы треугольников (рис. 7.7).

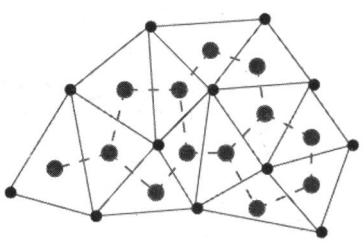


**Рис. 7.7.** Двойственный граф (пунктирные линии) триангулированной поверхности

Первым шагом в разработке программы, которая генерирует хороший набор полос треугольников, является создание двойственного графа триангулированной поверхности. Решение этой задачи я и поручил аспирантке. Через несколько дней она заявила, что одно лишь создание такого графа для поверхности из нескольких тысяч треугольников занимает на ее компьютере свыше пяти минут.

— Не может быть! — сказал я. — Ты, должно быть, строишь граф крайне нерационально. Какой формат данных ты используешь?

— Вначале идет список 3D-координат, используемых в модели вершин, а за ним — список треугольников. Каждый треугольник описывается списком из трех индексов координат вершин. Вот небольшой пример (листинг 7.6).

**Листинг 7.6. Пример описания треугольников**

```
VERTICES 4
0.000000 240.000000 0.000000
204.000000 240.000000 0.000000
204.000000 0.000000 0.000000
0.000000 0.000000 0.000000
TRIANGLES 2
0 1 3
1 2 3
```

— Понятно. Значит, первый треугольник использует все точки, кроме третьей, т. к. все индексы начинаются с нуля. Два треугольника должны иметь общую сторону, определяемую точками 1 и 3.

— Так оно и есть, — подтвердила она.

— Хорошо. Теперь расскажи мне, как ты строишь двойственный граф.

— Геометрическое расположение точек не влияет на структуру графа, поэтому я могу игнорировать информацию о вершинах. В моем двойственном графе будет такое же количество вершин, как и треугольников. Я создаю структуру данных в виде списка смежности, содержащего такое количество вершин. При считывании каждого треугольника я сравниваю его со всеми другими треугольниками, чтобы узнать, не имеют ли они две общие конечные точки. Если имеют, то я добавляю ребро между новым треугольником и этим.

Я не мог сдержать раздражения: «Но ведь именно *в этом* и заключается твоя проблема! Ты сравниваешь каждый треугольник со всеми остальными треугольниками, вследствие чего создание двойственного графа будет квадратичным по числу треугольников. Считывание входного графа должно занимать линейное время!»

— Я не сравниваю каждый треугольник со всеми другими треугольниками. В действительности в среднем он сравнивается с половиной или с третью других треугольников.

— Превосходно. Тем не менее у тебя алгоритм имеет квадратичную — $O(n^2)$ — сложность. Он слишком медленный.

Признавать свое поражение она не собиралась: «Вы только критикуете мой алгоритм, а можете ли Вы помочь мне исправить его?»

Вполне разумно. Я задумался. Нам был нужен какой-то быстрый способ отбросить большинство треугольников, которые не могли быть смежными с новым треугольником ($i, j, k$). Что нам в действительности было нужно, так это отдельный список всех треугольников, проходящих через точки $i$, $j$ и $k$. По формуле Эйлера для планарных графов средняя точка соединяет менее чем шесть треугольников. Это позволило бы сравнивать каждый новый треугольник менее чем с двадцатью другими треугольниками.

— Нам нужна структура данных, состоящая из массива элементов для каждой вершины в первоначальном наборе данных. Этим элементом будет список всех треугольников, которые проходят через ту или иную вершину. При считывании нового треугольника мы находим три соответствующих списка в массиве и сравниваем каждый из них с новым треугольником. Так что на самом деле придется проверить только два из этих трех списков, поскольку любые смежные треугольники будут иметь две общие точки. Нам надо будет добавить в наш граф ребро для каждой пары треугольников, имеющей две общие вершины. Наконец, новый треугольник добавляется в каждый из трех обработанных списков, чтобы они были текущими для считывания следующего треугольника.

Она немного поразмыслила над этим и улыбнулась: «Понятно. Я сообщу Вам о результатах».

На следующий день она доложила, что создание графа занимает несколько секунд даже для больших моделей. А затем написала хорошую программу для разбиения триангулированной поверхности на полосы треугольников, как описано в *разд. 3.6*.

#### Подведение итогов

Даже такие элементарные задачи, как инициализация структур данных, могут оказаться узким местом при разработке алгоритмов. Программы, работающие с большими объемами данных, должны иметь линейное или почти линейное время исполнения. Такие высокие требования к производительности не прощают небрежности. Сфокусировавшись на необходимости получения линейной производительности, вы наверняка сможете найти соответствующий детерминистический или эвристический алгоритм для решения поставленной задачи.

## 7.5. Обход графа

Самой фундаментальной задачей на графах, возможно, является систематизированное посещение каждой вершины и каждого ребра графа. Поэтому все основные служебные операции по работе с графами (такие как распечатка или копирование графов или преобразования графа из одного представления в другое) на самом деле являются приложениями обхода графа (graph traversal).

Лабиринты обычно представляются в виде графов, где вершины обозначают пересечения путей, а ребра — пути лабиринта. Таким образом, любой алгоритм обхода графа должен быть достаточно мощным, чтобы вывести нас из произвольного лабиринта. Чтобы обеспечить *эффективность* такого алгоритма, мы должны гарантировать, что не будем постоянно возвращаться в одну и ту же точку, оставаясь в лабиринте навечно. А для *правильности* нашего алгоритма нам нужно выполнять обход лабиринта систем-

ным образом, который гарантирует, что мы найдем выход из лабиринта. В поисках выхода нам нужно посетить каждую вершину и каждое ребро графа.

Ключевая идея обхода графа — пометить каждую вершину при первом ее посещении и помнить о том, что не было исследовано полностью. В сказках для обозначения пройденного пути использовались такие способы, как хлебные крошки и нитки, но в наших обходах графов мы воспользуемся булевыми флагами или перечислимыми типами.

Каждая вершина будет находиться в одном из следующих трех состояний:

- *неоткрытая* (undiscovered) — первоначальное, нетронутое состояние вершины;
- *открытая* (discovered) — вершина обнаружена, но мы еще не проверили все инцидентные ей ребра;
- *обработанная* (processed) — все инцидентные этой вершине ребра были посещены.

Очевидно, что вершину нельзя *обработать* до того, как она открыта, поэтому в процессе обхода графа состояние каждой вершины начинается с *неоткрытого*, переходит в *открытое* и заканчивается *обработанным*.

Нам также нужно иметь структуру, содержащую все открытые, но еще не обработанные вершины. Первоначально открытой считается только одна вершина — начало обхода графа. Для полного исследования вершины $v$ нужно изучить каждое исходящее из нее ребро. Если какие-либо ребра идут к неоткрытой вершине $x$, то эта вершина помечается как *открытая* и добавляется в список для дальнейшей обработки. Ребра, идущие к *обработанным* вершинам, игнорируются, т. к. их дальнейшее исследование не сообщит нам о графе ничего нового. По этой же причине можно игнорировать любое ребро, идущее к *открытой*, но не *обработанной* вершине, поскольку эта вершина уже внесена в список вершин, подлежащих обработке.

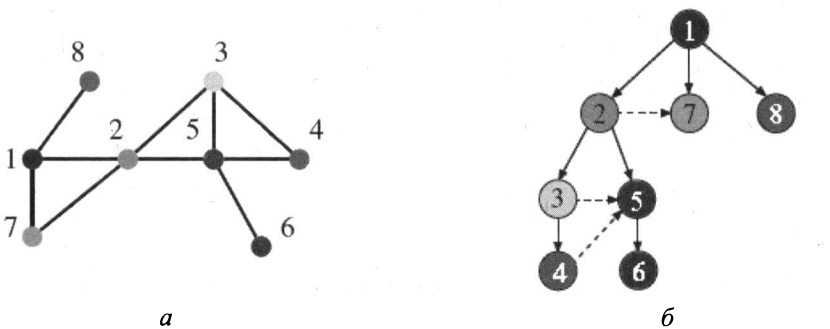
Каждое неориентированное ребро рассматривается дважды — по одному разу при исследовании каждой из его вершин. Ориентированные ребра рассматриваются только один раз — при исследовании его источника. В конечном счете все ребра и вершины в связном графе должны быть посещены. Почему? Допустим, что имеется непосещенная вершина $u$, чья соседняя вершина $v$ была посещена. Эта соседняя вершина $v$ будет со временем исследована, после чего мы непременно посетим вершину $u$. Таким образом, мы в конечном счете найдем все, что можно найти.

Далее мы обсудим механизм работы алгоритмов обхода графов и важность того, в каком порядке выполняется обход.

## 7.6. Обход в ширину

Обход в ширину (breadth-first traversal) является основой для многих важных алгоритмов для работы с графами. Далее приводится базовый алгоритм обхода графа в ширину (рис. 7.8). На определенном этапе каждая вершина графа переходит из состояния *неоткрытая* в состояние *открытая*. При обходе в ширину неориентированного графа каждому ребру присваивается направление: от открывающей вершины $u$ к открываемой вершине $v$. В этом контексте вершина $u$ называется родителем (parent), или предшественником (predecessor), вершины $v$, а вершина $v$ — потомком вершины $u$. По-

скольку каждый узел, за исключением корня, имеет только одного родителя, получится дерево вершин графа — оно и определяет кратчайший путь от корня ко всем другим узлам графа. Это свойство делает обход в ширину очень полезным в решении задач поиска кратчайшего пути. В листинге 7.7 приводится псевдокод алгоритма обхода графа в ширину.




*а* *б*

**Рис. 7.8.** Неориентированный граф (*а*) и его дерево обхода в ширину (*б*). Пунктирные линии не являются частью дерева, а обозначают ребра графа, идущие к открытым или обработанным вершинам

**Листинг 7.7. Обход графа в ширину**

```
BFS(G,s)

   Инициализируем каждую вершину u ∈ V [G] следующим образом:
      state[u] = "undiscovered"
      p[u] = nil, т. е. в начале дерева вершины-родители отсутствуют
   state[s] = "discovered"
   p[s] = nil
   Q = {s}
   while Q ≠ Ø do
      u = dequeue[Q]
      обрабатываем вершину u, если требуется
      для каждой вершины v, смежной с u
         обрабатываем ребро (u, v), если требуется
         if state[v] = "undiscovered" then
            state[v] = "discovered"
            p[v] = u
            enqueue[Q,v]
   state[u] = "processed"
```

Ребра графа, которые не включены в дерево обхода в ширину, также имеют особые свойства. Для неориентированных графов не попавшие в дерево ребра могут указывать только на вершины на том же уровне, что и родительская вершина, или на вершины, расположенные на уровень ниже. Эти свойства естественно следуют из того факта, что каждое ребро в дереве должно быть кратчайшим путем в графе. А для ориентированных графов ребро (*u*, *v*), указывающее в обратном направлении, может существовать в любом случае, когда вершина *v* расположена ближе к корню, чем вершина *u*.

### Реализация

Процедура обхода в ширину `bfs` использует два массива булевых значений для хранения информации о каждой вершине графа. Вершина *открывается* при первом ее посещении. Когда все исходящие из вершины ребра были исследованы, то вершина считается *обработанной*. Таким образом, как уже отмечалось ранее, в процессе обхода состояние каждой вершины начинается с *неоткрытого*, переходит в *открытое* и заканчивается *обработанным*. Эту информацию можно было бы хранить с помощью одной переменной перечислимого типа, но мы используем две булевы переменные:

```
bool processed[MAXV+1];          /* Обработанные вершины*/
bool discovered[MAXV+1];         /* Открытые вершины */
int parent[MAXV+1];              /* Отношения открытия */
```

Сначала каждая вершина инициализируется как неоткрытая (листинг 7.8).

**Листинг 7.8. Инициализация вершин**

```
void initialize_search(graph *g) {
    int i;                /* Счетчик */

    time = 0;

    for (i = 0; i <= g->nvertices; i++) {
        processed[i] = false;
        discovered[i] = false;
        parent[i] = -1;
    }
}
```

После открытия вершина помещается в очередь. Так как эти вершины обрабатываются в порядке FIFO, то первыми обрабатываются вершины, поставленные в очередь первыми, т. е. ближайшие к корню. Процедура обхода графа в ширину приводится в листинге 7.9.

**Листинг 7.9. Обход графа в ширину**

```
void bfs(graph *g, int start) {
    queue q;              /* Очередь вершин для обработки */
    int v;                /* Текущая вершина */
    int y;                /* Следующая вершина */
    edgenode *p;          /* Временный указатель */

    init_queue(&q);
    enqueue(&q, start);
    discovered[start] = true;

    while (!empty_queue(&q)) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
```

```
        p = g->edges[v];
        while (p != NULL) {
            y = p->y;
            if ((!processed[y]) || g->directed) {
                process_edge(v, y);
            }
            if (!discovered[y]) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

## 7.6.1. Применение обхода

Процедура обхода `bfs` использует функции `process_vertex_early()`, `process_vertex_late()` и `process_edge()`. С их помощью мы можем настроить действие, предпринимаемое этой процедурой при одноразовом посещении каждого ребра и вершины. Первоначально вся обработка вершин будет осуществляться при входе, поэтому функция `process_vertex_late()` не выполняет никаких действий (листинг 7.10).

**Листинг 7.10. Функция `process_vertex_late()`**

```
void process_vertex_late(int v) {

   /* Нет действий */
}
```

Следующие функции предназначены для вывода на экран (или принтер) каждой вершины и каждого ребра точно по одному разу (листинг 7.11).

**Листинг 7.11. Функции `process_vertex_early()` и `process_edge()`**

```
void process_vertex_early(int v) {
    printf("processed vertex %d\n", v);
}

void process_edge(int x, int y) {
    printf("processed edge (%d,%d)\n", x, y);
}
```

С помощью функции `process_edge()` можно также подсчитывать количество ребер (листинг 7.12).

**Листинг 7.12. Функция `process_edge()` для подсчета количества ребер**

```
void process_edge(int x, int y) {
    nedges = nedges + 1;
}
```

Разные алгоритмы предпринимают разные действия при посещении вершин и ребер. Использование этих функций предоставляет нам гибкость в настройке требуемого действия.

## 7.6.2. Поиск путей

Массив `parent`, который заполняется в процедуре `bfs`, очень полезен для поиска разнообразных путей в графе. Вершина, первая открывшая вершину $i$, определяется как `parent[i]`. Поскольку в процессе обхода все вершины открываются по одному разу, то каждая вершина, за исключением корневой, имеет родителя. Это родительское отношение определяет *дерево открытий*, корнем которого является первоначальный узел обхода.

Так как вершины открываются в порядке возрастающего расстояния от корня, то это дерево обладает очень важным свойством. Однозначно определенный путь от корня до каждой вершины $x \in V$ использует наименьшее количество ребер (или, что эквивалентно, промежуточных вершин), возможное в любом маршруте графа от корня до вершины $x$.

Этот путь можно воссоздать, следуя по цепи предшественников от вершины $x$ по направлению к корню. Обратите внимание, что в этом случае нам нужно двигаться в обратном направлении. Мы не можем найти путь от корня к вершине $x$, потому что указатели родителей имеют противоположное направление. Вместо этого путь и нужно искать по направлению от вершины $x$ к корню. Так как это направление противоположно тому, в котором требуется выполнять проход, мы можем либо сохранить путь, а потом явно обратить его с помощью стека, либо переложить эту работу на рекурсивную процедуру, приведенную в листинге 7.13.

**Листинг 7.13. Изменение направления пути посредством рекурсии**

```
void find_path(int start, int end, int parents[]) {
    if ((start == end) || (end == -1)) {
        printf("\n%d", start);
    } else {
        find_path(start, parents[end], parents);
        printf(" %d", end);
    }
}
```

При обходе в ширину графа, изображенного на рис. 7.8, наш алгоритм выдал следующие отношения «вершина/родитель»:

| Вершина | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Родитель | –1 | 1 | 2 | 3 | 2 | 5 | 1 | 1 |

Согласно этим родительским отношениям самый короткий путь от вершины 1 к вершине 6 проходит через набор вершин {1, 2, 5, 6}.

При использовании обхода в ширину для поиска кратчайшего пути от вершины $x$ к вершине $y$ нужно иметь в виду следующее: дерево кратчайшего пути полезно только в том случае, если корнем поиска в ширину является вершина $x$. Кроме того, поиск в ширину дает самый короткий путь только для невзвешенных графов. Алгоритмы поиска кратчайшего пути во взвешенных графах рассматриваются в *разд. 8.3.1*.

# 7.7. Применение обхода в ширину

Многие элементарные алгоритмы для работы с графами выполняют один или два обхода графа, в процессе которых они выполняют какие-либо действия. Любой из таких алгоритмов, если он корректно реализован с использованием списков смежности, обязательно имеет линейное время исполнения, поскольку обход в ширину исполняется за время $O(n + m)$ как на ориентированных, так и на неориентированных графах. Это оптимальное время, потому что именно за такое время можно лишь *прочитать* граф из $n$ вершин и $m$ ребер.

Секрет мастерства заключается в умении видеть ситуации, в которых применение таких обходов гарантированно даст положительные результаты. Далее приводится несколько примеров использования обхода в ширину.

## 7.7.1. Компоненты связности

Граф называется *связным* (connected), если имеется путь между любыми двумя его вершинами. Связность графа дружеских отношений означает, что любые два человека в нем связаны цепочкой из людей, попарно знакомых друг с другом.

*Компонентом связности* (connected component) неориентированного графа называется максимальный набор его вершин, для которого существует путь между каждой парой вершин. Эти компоненты являются отдельными «кусками» графа, которые не соединены между собой. В качестве примера отдельных компонентов связности в графе дружеских отношений можно привести обитающие где-то в джунглях первобытные племена, которые еще не были открыты для остального мира. А отшельник в пустыне или крайне неприятный человек будет примером компонента связности, состоящего из одной вершины.

Удивительно, какое большое количество кажущихся сложными проблем сводится к поиску или подсчету компонентов связности. Например, вопрос, можно ли решить какую-нибудь головоломку (скажем, кубик Рубика), начав с определенной позиции, по сути, представляет собой вопрос, является ли связным граф возможных конфигураций.

Компоненты связности можно найти с помощью обхода в ширину, т. к. порядок перечисления вершин не имеет значения. Начнем с выполнения поиска, производя его от

произвольной вершины. Все элементы, обнаруженные в процессе этого обхода, должны быть членами одного и того же компонента связности. Потом повторим обход, начиная с любой неоткрытой вершины (если таковая имеется), чтобы определить второй компонент связности, — и т. д. до тех пор, пока не будут обнаружены все вершины. Соответствующая процедура приводится в листинге 7.14.

**Листинг 7.14. Процедура поиска компонентов связности**

```
void connected_components(graph *g) {
   int c;              /* Номер компонента */
    int i;             /* Счетик */

    initialize_search(g);

    c = 0;
    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            c = c + 1;
            printf("Component %d:", c);
            bfs(g, i);
            printf("\n");
        }
    }
}

void process_vertex_early(int v) {  /* Вершина для обработки */
    printf(" %d", v);
}

void process_edge(int x, int y) {
}
```

Обратите внимание на увеличение значения счетчика `c`, содержащего номер текущего компонента, при каждом вызове функции `bfs`. Альтернативно, изменив соответствующим образом действие функции `process_vertex()`, каждую вершину можно было бы явно связать с номером ее компонента (вместо того, чтобы выводить на экран вершины каждого компонента).

Для ориентированных графов существуют два понятия связности, и это определяет существование алгоритмов поиска компонентов сильной и слабой связности. Любой из них можно найти за время $O(n + m)$, что показано в *разд. 18.1*.

## 7.7.2. Раскраска графов двумя цветами

В задаче *раскраски вершин* (vertex colouring) требуется присвоить метку (или цвет) каждой вершине графа таким образом, чтобы любые две соединенные ребром вершины были разного цвета. Присвоение каждой вершине своего цвета позволяет избежать любых конфликтов. Но при этом нужно использовать как можно меньшее количество цветов. Задачи раскраски вершин часто возникают в приложениях календарного пла-

нирования — например, при выделении регистров в компиляторах. Подробное обсуждение алгоритмов раскраски вершин можно найти в *разд. 19.7*.

Граф называется *двудольным* (bipartite), если его вершины можно правильно раскрасить двумя цветами. Важность двудольных графов заключается в том, что они возникают естественным образом во многих приложениях. Рассмотрим граф взаимных заинтересованностей среди гетеросексуалов, где люди рассматривают в качестве возможных партнеров только лиц противоположного пола. В этой простой модели двухцветная раскраска графа будет определяться полом.

Но как мы можем правильно раскрасить такой граф в два цвета, разделив таким образом мужчин и женщин? Для этого мы, например, объявим в приказном порядке, что начальная вершина обхода представляет мужчину. Тогда все смежные вершины должны представлять женщин — при условии, что граф в действительности является двудольным.

Мы можем расширить алгоритм обхода в ширину таким образом, чтобы раскрашивать каждую новую открытую вершину цветом, противоположным цвету ее предшественника. Потом мы проверяем, не связывает какое-либо ребро, которое не входит в дерево, две вершины одного цвета. Наличие такой конфликтной связи будет означать, что граф нельзя раскрасить в два цвета. Отсутствие этих конфликтов по окончании процесса означает, что мы создали правильную двухцветную раскраску графа. Процедура двухцветной раскраски графа приведена в листинге 7.15.

**Листинг 7.15. Процедура раскраски графов двумя цветами**

```
void twocolor(graph *g) {
    int i;        /* Счетчик */

    for (i = 1; i <= (g->nvertices); i++) {
        color[i] = UNCOLORED;
    }

    bipartite = true;

    initialize_search(g);

    for (i = 1; i <= (g->nvertices); i++) {
        if (!discovered[i]) {
            color[i] = WHITE;
            bfs(g, i);
        }
    }
}

void process_edge(int x, int y) {
    if (color[x] == color[y]) {
        bipartite = false;
        printf("Warning: not bipartite, due to (%d,%d)\n", x, y);
    }
```

```
    color[y] = complement(color[x]);
}

int complement(int color) {
    if (color == WHITE) {
        return(BLACK);
    }

    if (color == BLACK) {
        return(WHITE);
    }

return(UNCOLORED);
}
```

Первой вершине в любом компоненте связности можно присвоить любой цвет (пол). Алгоритм обхода в ширину может разделить мужчин и женщин, но, исходя только из структуры графа, не может определить, какой цвет представляет какой пол. Кроме этого, двудольные графы требуют различных и двоичных атрибутов категории, поэтому они не моделируют реальное варьирование в половых предпочтениях и половой самоидентификации.

#### Подведение итогов

Обходы в ширину и в глубину (см. далее) предоставляют механизмы для посещения каждой вершины и каждого ребра графа. Они лежат в основе большинства простых и эффективных алгоритмов для работы с графами.

## 7.8. Обход в глубину

Существуют два основных алгоритма обхода графов: *обход в ширину* (breadth-first search, BFS) и *обход в глубину* (depth-first search, DFS). Для некоторых задач нет абсолютно никакой разницы, какой тип обхода (поиска) использовать, но для других эта разница является критической.

Разница между поиском в ширину и поиском в глубину заключается в порядке исследования вершин. Этот порядок зависит полностью от структуры-контейнера, используемой для хранения *открытых*, но не *обработанных* вершин.

- *Очередь.*

  Помещая вершины в очередь типа FIFO, мы исследуем самые старые неисследованные вершины первыми. Таким образом, наше исследование медленно распространяется вширь, начиная от стартовой вершины. В этом суть обхода в ширину.

- *Стек.*

  Помещая вершины в стек с порядком извлечения LIFO, мы исследуем их, отклоняясь от пути для посещения очередного соседа, если таковой имеется, и возвращаясь назад, только если оказываемся в окружении ранее открытых вершин. Таким образом, мы в своем исследовании быстро удаляемся от стартовой вершины, и в этом заключается суть обхода в глубину.

Наша реализация процедуры обхода в глубину отслеживает *время* обхода для каждой вершины. Каждый вход в вершину и выход из нее считаются затратой времени и инкрементируют счетчик `time`. Для каждой вершины ведется учет затрат времени на *вход* и *выход*.

Процедуру обхода в глубину можно реализовать рекурсивным методом, что позволяет избежать явного использования стека. Псевдокод алгоритма обхода в глубину приведен в листинге 7.16.

**Листинг 7.16. Обход в глубину**

```
DFS(G, u)
    state[u] = "discovered"
    обрабатываем вершину u, если необходимо
    time = time + 1
    entry[u] = time
    для каждой вершины v, смежной с вершиной u
        обрабатываем ребро (u, v), если необходимо
        если state[v] = "undiscovered", тогда
            p[v] = u
            DFS(G, v)
    state[u] = "processed"
    exit[u] = time
    time = time + 1
```

При обходе в глубину интервалы времени, потраченного на посещение вершин, обладают интересными свойствами. В частности:

- *Посещение предшественника.*

  Допустим, что вершина $x$ является предшественником вершины $y$ в дереве обхода в глубину. Это подразумевает, что вершина $x$ должна быть посещена раньше, чем вершина $y$, т. к. невозможно быть рожденным раньше своего отца или деда. Кроме этого, выйти из вершины $x$ можно только после выхода из вершины $y$, поскольку механизм поиска в глубину предотвращает выход из вершины $x$ до тех пор, пока мы не вышли из всех ее потомков. Таким образом, временной интервал посещения $y$ должен быть корректно размещен в интервале его предшественника $x$.

- *Количество потомков.*

  Разница во времени выхода и входа для вершины $v$ свидетельствует о количестве потомков этой вершины в дереве обхода в глубину. Показания часов увеличиваются на единицу при каждом входе и каждом выходе из вершины, поэтому количество потомков той или иной вершины $v$ будет равно половине разности между моментом выхода и моментом входа.

Мы будем использовать время входа и выхода в разных приложениях обхода в глубину — в частности, в топологической сортировке и в поиске компонентов двусвязности (или компонентов сильной связности). Нам, возможно, нужно будет выполнять разные действия при каждом входе и выходе из вершины, для чего из процедуры `dfs` будут вызываться функции `process_vertex_early()` и `process_vertex_late()` соответственно.

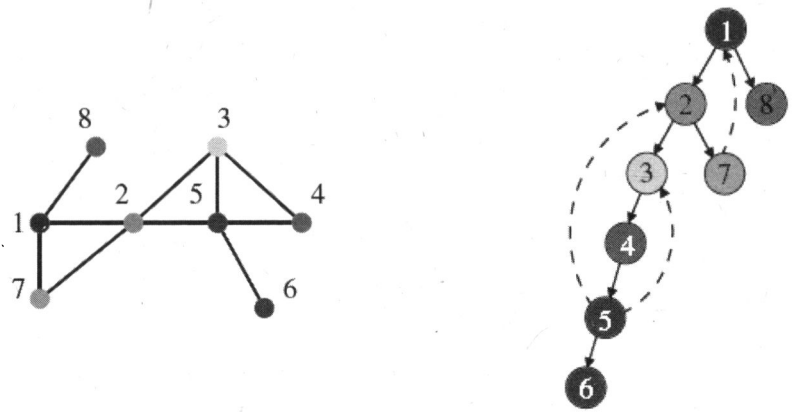



**Рис. 7.9.** Неориентированный граф (*а*) и его дерево обхода в глубину (*б*). Пунктирные линии не являются частью дерева, а лишь обозначают обратные ребра

На рис. 7.9 приводится пример неориентированного графа (*а*) и дерева его обхода в глубину (*б*).

Родители каждой вершины дерева, приведенного на рис. 7.9, *б*, и время входа и выхода из нее определяются следующими соотношениями:

| Вершина | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Родитель | –1 | 1 | 2 | 3 | 4 | 5 | 2 | 1 |
| Время входа | 1 | 2 | 3 | 4 | 5 | 6 | 11 | 14 |
| Время выхода | 16 | 13 | 10 | 9 | 8 | 7 | 12 | 15 |

Другим важным свойством обхода в глубину является то, что он разбивает ребра неориентированного графа на два класса: *древесные* (tree edges) и *обратные* (back edges). Древесные ребра используются при открытии новых вершин и закодированы в родительском отношении. Обратные ребра — это те ребра, у которых второй конец является предшественником расширяемой вершины, и поэтому они направлены обратно к дереву.

Удивительным свойством обхода в глубину является то, что все ребра попадают в одну из этих двух категорий. Почему ребро не может соединять одноуровневые узлы, а только родителя с потомком? Потому что все вершины, достижимые из той или иной вершины $v$, уже исследованы ко времени окончания обхода, начатого из вершины $v$, и такая топология невозможна для неориентированных графов. Приведенная классификация ребер является принципиальной для алгоритмов, основанных на обходе в глубину.

### Реализация

Обход в глубину можно рассматривать как обход в ширину с использованием стека вместо очереди для хранения необработанных вершин. Достоинством реализации об-

хода в глубину посредством рекурсии (листинг 7.17) является то, что она позволяет обойтись без явного использования стека.

**Листинг 7.17. Обход в глубину**

```
void dfs(graph *g, int v) {
    edgenode *p;            /* Временный указатель */
    int y;                  /* Следующая вершина */

    if (finished) {
        return;             /* Завершение поиска */
    }
    discovered[v] = true;
    time = time + 1;
    entry_time[v] = time;

    process_vertex_early(v);

    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (!discovered[y]) {
            parent[y] = v;
            process_edge(v, y);
            dfs(g, y);
        } else if ((!processed[y]) && (parent[v] != y)) || (g->directed)) {
            process_edge(v, y);
        }

        if (finished) {
            return;
        }
        p = p->next;
    }

    process_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;
    processed[v] = true;
}
```

Обход в глубину, по сути, эксплуатирует ту же самую идею, что и *поиск с возвратом*, который рассматривается в *разд. 9.1*. В обоих алгоритмах мы перебираем все возможности, продвигаясь вперед, когда это удается, и возвращаясь обратно только тогда, когда больше не осталось неисследованных элементов для дальнейшего продвижения. Оба алгоритма легче всего воспринимать как рекурсивные алгоритмы.

### Подведение итогов

При обходе в глубину вершины упорядочиваются по времени входа/выхода, а ребра разбиваются на два типа: древесные и обратные. Именно такая организация и делает обход в глубину столь мощным алгоритмом.

# 7.9. Применение обхода в глубину

По сравнению с другими парадигмами разработки алгоритмов обход в глубину не кажется чем-то страшным. Но он весьма сложен, вследствие чего для его корректной работы необходимо правильно реализовать все до мельчайших подробностей.

Правильность алгоритма обхода в глубину зависит от того, *когда* именно обрабатываются ребра и вершины. Вершину $v$ можно обработать до обхода исходящих из нее ребер (процедура `process_vertex_early()`) или после этого (процедура `process_vertex_late()`). Иногда действия выполняются в обоих случаях — например, инициализация до обхода ребер посредством функции `process_vertex_early()` специфической структуры данных обрабатываемых вершин, которая будет модифицирована операциями обработки ребер, а после обхода подвергнута анализу посредством функции `process_vertex_late()`.

В неориентированных графах каждое ребро $(x, y)$ находится в списке смежности и для вершины $x$, и для вершины $y$. Таким образом, для обработки каждого ребра $(x, y)$ существуют два момента: при исследовании вершины $x$ и вершины $y$. Ребра разбиваются на древесные и обратные при первом исследовании ребра. Момент, в который мы видим ребро в первый раз, естественно подходит для выполнения специфической обработки ребра. В тех случаях, когда мы встречаемся с ребром во второй раз, может возникнуть необходимость в каком-то другом действии.

Но если мы встречаем ребро $(x, y)$, двигаясь из вершины $x$, как нам определить, не было ли это ребро уже пройдено из вершины $y$? На этот вопрос легко ответить, если вершина $y$ неоткрытая — ребро $(x, y)$ становится древесным ребром, следовательно, мы видим его в первый раз. Ответ также очевиден, если вершина $y$ была полностью обработана — поскольку мы рассматривали ребро при исследовании вершины $y$, то это второе посещение этого ребра. А если вершина $y$ является предшественником вершины $x$ и вследствие этого находится в открытом состоянии? Подумав, мы поймем, что это должен быть первый проход, *если только* вершина $y$ не является непосредственным предшественником вершины $x$, т. е. ребро $(y, x)$ является древесным. Это можно установить проверкой равенства `y == parent[x]`.

Каждый раз, когда я пробую реализовать алгоритм на основе обхода в глубину, я убеждаюсь, что этот тип алгоритмов достаточно сложен[2]. Поэтому я рекомендую вам тщательно изучить рассмотренные реализации этого алгоритма, чтобы понять, где и почему могут возникать проблемы.

## 7.9.1. Поиск циклов

Наличие обратных ребер является ключевым фактором при поиске циклов в неориентированных графах. Если в графе нет обратных ребер, то все ребра являются древесными и дерево не содержит циклов. Но *любое* обратное ребро, идущее от вершины $x$ к предшественнику $y$, создает цикл или замкнутый маршрут между вершинами $y$ и $x$. Цикл легко найти посредством обхода в глубину, как показано в листинге 7.18.

---

[2] Собственно говоря, самые ужасные ошибки в предыдущем издании этой книги были как раз в этом разделе.

**Листинг 7.18. Поиск цикла**

```
void process_edge(int x, int y) {
    if (parent[y] != x) { /* Найдено обратное ребро! */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        finished = true;
    }
}
```

Правильность этого алгоритма поиска цикла зависит от обработки каждого неориентированного ребра только один раз. В противном случае двойной проход по любому неориентированному ребру может создать фиктивный двухвершинный цикл $(x, y, x)$. Поэтому после обнаружения первого цикла устанавливается флаг `finished`, что ведет к завершению работы процедуры. Без этого флага мы бы тратили понапрасну время, открывая новый цикл с каждым обратным ребром, прежде чем остановиться, — не забудем, что полный граф содержит $\Theta(n^2)$ таких циклов.

## 7.9.2. Шарниры графа

Допустим, вы — диверсант в тылу врага, которому нужно вывести из строя телефонную сеть противника. Какую из показанных на рис. 7.10 телефонных станций вы бы решили взорвать, чтобы причинить как можно больший ущерб?

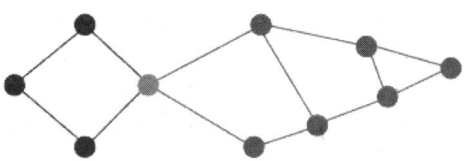


**Рис. 7.10.** Граф телефонной сети врага, которую нужно вывести из строя

Обратите внимание, что в графе, показанном на рис. 7.10, имеется одна точка, удаление которой отсоединяет связный компонент графа. Такая вершина называется *шарниром* (или *разделяющей вершиной*, или *точкой сочленения*). Любой граф, содержащий шарнир, является уязвимым по своей природе, поскольку удаление этой одной вершины ведет к потере связности между другими вершинами. В *разд. 7.7.1* мы рассмотрели алгоритм на основе поиска в ширину для определения связных компонентов графа. В общем, *связностью* графа (graph connectivity) называется наименьший набор вершин, в результате удаления которых граф станет несвязным. Если граф имеет шарнир, то его связность равна единице. Более устойчивые графы, в которых шарниры отсутствуют, называются *двусвязными*. Связность также рассматривается более подробно в *разд. 18.8*.

Шарниры графа можно с легкостью найти простым перебором. Для этого мы временно удаляем вершину $v$, после чего выполняем обход оставшегося графа в ширину или в глубину, чтобы выяснить, остался ли он после этого связным. Эта процедура повторяется для каждой вершины графа. Общее время для $n$ таких обходов равно $O(n(m + n))$. Но существует изящный алгоритм с линейным временем исполнения, который проверяет все вершины связного графа за один обход в глубину.

Какую информацию о шарнирах может предоставить нам дерево обхода в глубину? Прежде всего, это дерево содержит все вершины связного графа (рис. ЦВ-7.11, *а*). И если дерево обхода в глубину представляет весь граф, то все его внутренние (не листовые) вершины будут шарнирами, т. к. удаление любой из них отделит лист от корня. А вот удаление какого-либо листа (обозначены зеленым на рис. ЦВ-7.11, *б*) разъединить дерево не может, поскольку листовая вершина не соединяет с деревом никакие другие вершины.

Корень дерева обхода представляет специальный случай. Если он имеет только одного потомка, то корень также является листом. Но если корень имеет больше одного потомка, то удаление корня разделяет потомков, а это означает, что корень является шарниром.

Общие графы более сложные, чем деревья. Но обход общего графа в глубину разделяет ребра на древесные и обратные. Обратные ребра можно рассматривать как страховочные фалы, связывающие вершину с одним из ее предшественников. Например, наличие обратного ребра от вершины $x$ к вершине $y$ гарантирует, что ни одна из вершин на пути между $x$ и $y$ не может быть шарниром. Удаление любой из этих вершин не нарушит связности дерева, потому что обратное ребро — как страховочный фал — будет удерживать их связанными с остальными компонентами дерева.

При поиске шарниров графа требуется отслеживать информацию о том, в какой степени обратные ребра связывают части дерева обхода в глубину после возможного шарнира с предшествующими узлами. Пусть переменная `reachable_ancestor[v]` обозначает самого старшего предшественника вершины $v$, достижимого с потомка вершины $v$ через обратное ребро. Сначала значение этой переменной равно `reachable_ancestor[v]=v` (листинг 7.19).

**Листинг 7.19. Инициализация массива достижимых предшественников**

```
int reachable_ancestor[MAXV+1];  /* Самый старший достижимый
                                    предшественник вершины v */
int tree_out_degree[MAXV+1];     /* Степень исхода вершины v
                                      дерева обхода в глубину*/
void process_vertex_early(int v) {
    reachable_ancestor[v] = v;
}
```

Переменная `reachable_ancestor[v]` обновляется при каждом обнаружении обратного ребра, которое ведет к более старшему предшественнику, чем текущий. Относительный возраст (ранг) предшественников можно определить по значению переменной `entry_time`, содержащей время входа в вершину (листинг 7.20).

**Листинг 7.20. Определение возраста предшественников**

```
void process_edge(int x, int y) {
    int class; /* edge class */

    class = edge_classification(x, y);
```

```
    if (class == TREE) {
        tree_out_degree[x] = tree_out_degree[x] + 1;
    }

    if ((class == BACK) && (parent[x] != y)) {
        if (entry_time[y] < entry_time[reachable_ancestor[x]]) {
            reachable_ancestor[x] = y;
        }
    }
}
```

Здесь исключительно важно определить, как достижимость влияет на то, является ли вершина $v$ шарниром. Существуют три типа шарниров, как показано на рис. 7.12. Обратите внимание на то, что эти типы шарниров не являются взаимоисключающими. Одиночная вершина $v$ может быть шарниром по нескольким причинам:

- *Корневые шарниры.*

  Если корень дерева обхода в глубину имеет свыше одного потомка, то он должен быть шарниром, т. к. при его удалении никакое ребро не может соединить поддерево второго потомка с поддеревом первого потомка.

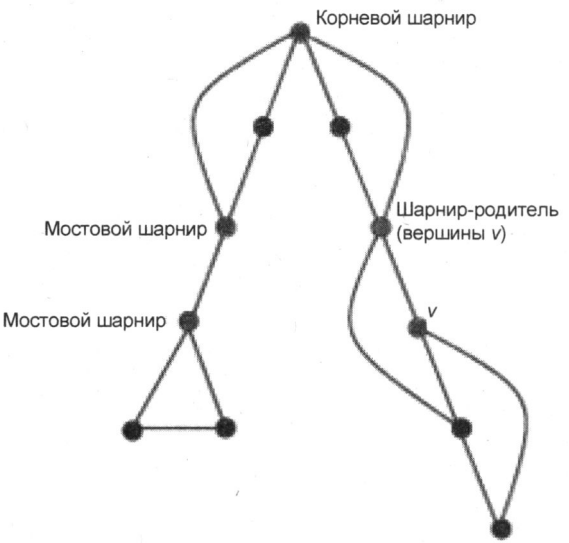



**Рис. 7.12.** Типы шарниров

- *Мостовые шарниры.*

  Если самой первой достижимой из вершины $v$ является вершина *родитель*[$v$], то удаление ребра *родитель*[$v$], $v$ разрывает граф. Очевидно, что вершина *родитель*[$v$] должна быть шарниром, т. к. ее удаление отсоединяет вершину $v$ от остального дерева обхода графа в глубину. По этой же причине шарниром является и вершина $v$, если только она не является листом дерева обхода в глубину. Удаление любого листа удаляет только сам лист, но ничего после него.

- *Родительские шарниры.*

  Если самой первой вершиной, достижимой из вершины *v*, является родитель вершины *v*, то удаление этого родителя должно отрезать вершину *v* от остальной части дерева, если только такой родитель не является корнем дерева. Это всегда происходит с более глубокой вершиной моста, если только она не является листом.

В листинге 7.21 приводится процедура для определения соответствия вершины, из которой выполняется возврат после обхода всех ее исходящих ребер, одному из этих трех типов шарниров.

«Возраст» вершины *v* определяется переменной `entry_time[v]`. Вычисляемое время `time_v` обозначает самую старшую вершину, достигаемую посредством обратных ребер. Возвращение к предшественнику выше вершины *v* исключает возможность того, что вершина *v* может быть шарниром.

**Листинг 7.21. Определение типа шарнира**

```
void process_vertex_late(int v) {
    bool root;          /* Эта вершина[v] — корень дерева обхода в глубину? */
    int time_v;         /* Самое раннее время достижимости вершины  v */
    int time_parent;    /* Самое раннее время достижимости вершины parent[v] */

    if (parent[v] == -1) { /* Вершина v — корень? */
        if (tree_out_degree[v] > 1) {
            printf("root articulation vertex: %d \n",v);
                  // Корневой шарнир
        }
        return;
    }

    root = (parent[parent[v]] == -1); /* Вершина parent[v] — корень? */

    if (!root) {
        if (reachable_ancestor[v] == parent[v]) {
            printf("parent articulation vertex: %d \n", parent[v]);
                 // Шарнир-родитель:
        }

        if (reachable_ancestor[v] == v) {
            printf("bridge articulation vertex: %d \n",parent[v]);

                if (tree_out_degree[v] > 0) { /* Вершина v — лист? */
                    printf("bridge articulation vertex: %d \n", v);
                         // Мостовой шарнир:

            }
        }
    }

    time_v = entry_time[reachable_ancestor[v]];
    time_parent = entry_time[reachable_ancestor[parent[v]]];
```

```
    if (time_v < time_parent) {
        reachable_ancestor[parent[v]] = reachable_ancestor[v];
    }
}
```

В последних строчках кода этой процедуры определяется момент, когда мы возвращаемся из самого старого достижимого предшественника вершины к ее родителю, а именно тот момент, когда он выше, чем текущий наивысший предшественник родителя.

В качестве альтернативы уязвимость можно рассматривать не с точки зрения слабости вершин, а с точки зрения слабости ребер. Возможно, нашему диверсанту вместо вывода из строя телефонной станции было бы легче повредить кабель. Ребро, чье удаление разъединяет граф, называется *мостом* (bridge). Граф, не имеющий мостов, называется *реберно-двусвязным* (edge-biconnected).

Является ли мостом то или иное ребро $(x, y)$, можно с легкостью определить за линейное время, удалив это ребро и проверив, является ли получившийся граф связным. Более того, можно найти все мосты за такое же линейное время $O(n + m)$ посредством обхода графа в глубину. Ребро $(x, y)$ является мостом, если, во-первых, оно древесное, а во-вторых, нет обратных ребер, которые соединяли бы вершину $y$ или нижележащие вершины с вершиной $x$ или вышележащими вершинами. Соответствие ребра этим условиям можно проверить с помощью модифицированной должным образом функции `process_vertex_late` (см. листинг 7.21).

## 7.10. Обход в глубину ориентированных графов

Обход в глубину неориентированного графа полезен тем, что он аккуратно упорядочивает ребра графа. В процессе выполнения обхода графа в глубину с заданной исходной вершины каждое ребро может быть отнесено к одному из четырех типов (рис. 7.13).

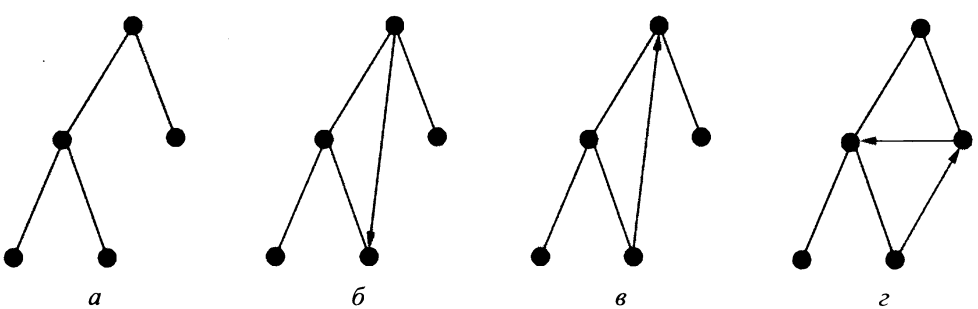



**Рис. 7.13.** Возможные типы ребер при обходе графа: *а* — древесные ребра; *б* — прямое ребро; *в* — обратное ребро; *г* — поперечные ребра. Прямые и поперечные ребра могут встречаться при обходе в глубину *только* в ориентированных графах

При обходе *неориентированных* графов каждое ребро или находится в дереве обхода в глубину, или будет обратным ребром к предшествующему ребру в дереве. Важно знать, почему это так. Возможно ли при обходе выйти на прямое ребро $(x, y)$, направ-

ленное к вершине-потомку? Нет, ибо в таком случае мы бы уже прошли это ребро $(x, y)$ при исследовании вершины $y$, что делает его обратным ребром. Возможно ли выйти на поперечное ребро $(x, y)$, связывающее две вершины, не имеющие отношения друг к другу? Опять же, нет, т. к. мы бы уже открыли это ребро при исследовании вершины $y$, что делает его древесным ребром.

Но для *ориентированных графов* диапазон допустимых типов ребер обхода в глубину можно расширить на все четыре типа ребер, показанных на рис. 7.13. Поэтому такая их классификация оказывается полезной при разработке алгоритмов для работы с ориентированными графами, поскольку для каждого типа ребра обычно предпринимается соответствующее ему действие.

Тип ребра можно без труда определить исходя из состояния вершины, времени ее открытия и ее родителя. Соответствующая процедура приведена в листинге 7.22.

**Листинг 7.22. Определение типа ребра**

```
int edge_classification(int x, int y) {
    if (parent[y] == x) {
        return(TREE);
    }

    if (discovered[y] && !processed[y]) {
        return(BACK);
    }

    if (processed[y] && (entry_time[y]>entry_time[x])) {
        return(FORWARD);
    }

    if (processed[y] && (entry_time[y]<entry_time[x])) {
        return(CROSS);
    }

    printf("Warning: self loop (%d,%d)\n", x, y);

    return -1;
}
```

Точно так же, как и в случае с процедурой обхода в ширину, реализация алгоритма обхода в глубину содержит места, в которые можно вставить функцию выборочной обработки каждой вершины или ребра — например, копировать, выводить на экран или подсчитывать их. Оба алгоритма обхода, как в глубину, так и в ширину, начинают с обхода всех ребер в одном компоненте связности. Для обхода несвязного графа они оба должны начинать с какой-либо вершины в каждом компоненте. Единственное существенное различие между ними состоит в том, как они упорядочивают и помечают ребра.

Я рекомендую читателям проверить мои рассуждения и убедиться в правильности четырех состояний, рассмотренных ранее. Все, сказанное мною о сложности алгоритма обхода в глубину, вдвойне справедливо для ориентированных графов.

## 7.10.1. Топологическая сортировка

Топологическая сортировка является наиболее важной операцией на бесконтурных ориентированных графах. Этот тип сортировки упорядочивает вершины вдоль линии таким образом, чтобы все ориентированные ребра были направлены слева направо. Такое упорядочивание ребер невозможно в графе, содержащем ориентированный цикл, поскольку нельзя сделать так, чтобы ребра шли по одной линии в одну сторону и при этом возвращались назад в исходную точку.

Любой бесконтурный ориентированный граф имеет, по крайней мере, одно топологическое упорядочивание (рис. 7.14).

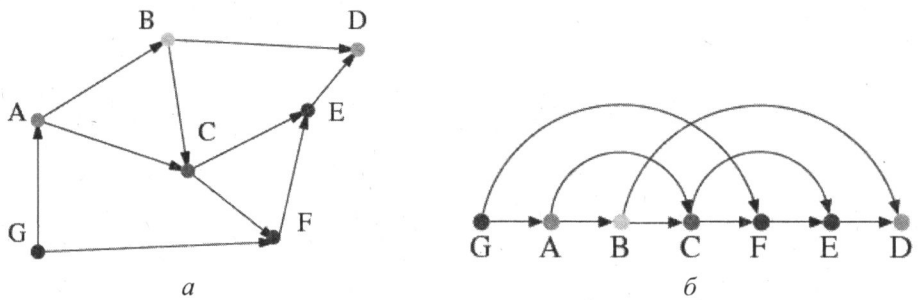



**Рис. 7.14.** Бесконтурный ориентированный граф (*а*)
с единственным топологическим упорядочиванием: G, A, B, C, F, E, D (*б*)

Важность топологической сортировки состоит в том, что она позволяет упорядочить вершины графа таким образом, что каждую вершину можно обработать перед обработкой ее потомков. Допустим, что ориентированные ребра представляют управление очередностью таким образом, что ребро $(x, y)$ означает, что работу $x$ нужно выполнить раньше, чем работу $y$. Тогда любое топологическое упорядочивание определяет возможное календарное расписание. Более того, бесконтурный ориентированный граф может содержать несколько таких упорядочиваний.

Но топологическая сортировка имеет и другие применения. Например, предположим, что нам нужно найти самый короткий (или самый длинный) путь в бесконтурном ориентированном графе от точки $x$ до точки $y$. Никакая вершина $v$, расположенная в топологическом упорядочивании после вершины $y$, не может находиться в этом пути, поскольку из такой вершины $v$ будет невозможно попасть обратно в вершину $y$. Все вершины можно обработать должным образом слева направо в топологическом порядке, принимая во внимание влияние их исходящих ребер и будучи в полной уверенности, что мы просмотрим все, что нужно, перед тем как оно потребуется. Топологическая сортировка оказывается очень полезной для решения практически любых алгоритмических задач по бесконтурным ориентированным графам, что подробно обсуждается в *разд. 18.2*.

Обход в глубину является эффективным средством для осуществления топологической сортировки. Ориентированный граф является бесконтурным, если не содержит обратных ребер. Топологическое упорядочивание бесконтурного ориентированного графа осуществляется посредством маркировки вершин в порядке, обратном тому, в котором

они отмечаются как *обработанные*. Почему это возможно? Рассмотрим, что происходит с каждым ориентированным ребром $(x, y)$, обнаруженным при исследовании вершины $x$:

- если вершина $y$ *не открыта*, то мы начинаем обход в глубину из вершины $y$, прежде чем сможем продолжать исследование вершины $x$. Таким образом, вершина $y$ должна быть помечена как *обработанная* до того, как такой статус присваивается вершине $x$, и поэтому в топологическом упорядочивании вершина $x$ будет находиться перед вершиной $y$, что и требуется;
- если вершина $y$ *открыта*, но не *обработана*, то ребро $(x, y)$ является обратным ребром, что невозможно в бесконтурном ориентированном графе, поскольку это создает цикл;
- если вершина $y$ *обработана*, то она помечается соответствующим образом раньше вершины $x$. Следовательно, в топологическом упорядочивании вершина $x$ будет находиться перед вершиной $y$, что и требуется.

В листинге 7.23 приводится реализация топологической сортировки.

**Листинг 7.23. Топологическая сортировка**

```
void process_vertex_late(int v) {
    push(&sorted, v);
}

void process_edge(int x, int y) {
    int class;              /* Класс ребра */

    class = edge_classification(x, y);

    if (class == BACK) {
        printf("Warning: directed cycle found, not a DAG\n");
                // Внимание: обнаружен ориентированный цикл,
                // неориентированный граф
    }
}

void topsort(graph *g) {
    int i;                  /* Счетчик */

    init_stack(&sorted);

    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }
    print_stack(&sorted);       /* Выводим топологическое упорядочивание */
}
```

Здесь каждая вершина помещается в стек после обработки всех исходящих ребер. Самая верхняя вершина в стеке не имеет входящих ребер, идущих от какой-либо вер-

шины, имеющейся в стеке. Последовательно снимая вершины со стека, получаем их топологическое упорядочивание.

## 7.10.2. Сильно связные компоненты

Ориентированный граф является *сильно связным* (strongly connected), если для любой пары его вершин ($v$, $u$) вершина $v$ достижима из вершины $u$ и наоборот. В качестве практического примера сильно связного графа можно привести граф дорожных сетей с двусторонним движением.

Является ли граф $G = (V, E)$ сильно связным, можно с легкостью проверить посредством обхода графа за линейное время. Граф $G$ является сильно связным тогда и только тогда, когда для любой вершины $v$ в $G$:

1. Все вершины графа $G$ достижимы из вершины $v$.
2. Вершина $v$ достижима из всех вершин в графе $G$.

Проверить действительность первого условия можно, выполнив обход графа в ширину или глубину из вершины $v$, чтобы установить, можно ли открыть все вершины. Если да, то тогда они все должны быть достижимы из вершины $v$.

Чтобы проверить достижимость вершины $v$ из всех других вершин графа, создадим транспонированный граф $G^T = (V, E')$, содержащий такое же множество вершин и ребер, как и граф G, но с инвертированными ребрами. Иными словами, ориентированное ребро $(x, y) \in E$ тогда и только тогда, когда $(y, x) \in E'$. Соответствующий код приводится в листинге 7.24.

**Листинг 7.24. Транспонирование графа**

```
graph *transpose(graph *g) {
    graph *gt; /* транспонирование графа g */
    int x; /* counter */
    edgenode *p; /* временный указатель */

    gt = (graph *) malloc(sizeof(graph));
    initialize_graph(gt, true); /* инициализировать ориентированный граф */
    gt->nvertices = g->nvertices;

    for (x = 1; x <= g->nvertices; x++) {
        p = g->edges[x];
        while (p != NULL) {
            insert_edge(gt, p->y, x, true);
            p = p->next;
        }
    }

    return(gt);
}
```

Любой путь от вершины $v$ к вершине $z$ в графе $G^T$ соответствует пути от вершины $z$ к вершине $v$ в графе $G$. Выполнив еще один обход графа в глубину, но на этот раз из

вершины v в графе $G^T$, мы определим все вершины, достижимые из вершины $v$ в графе $G$.

Все ориентированные графы можно разбить на *сильно связные компоненты* таким образом, чтобы каждая пара вершин того или иного компонента была соединена ориентированным путем, как показано на рис. 7.15, *а*. Множество таких компонентов можно определить при помощи более тонкого варианта обхода графа в глубину, код которого приводится в листинге 7.25.

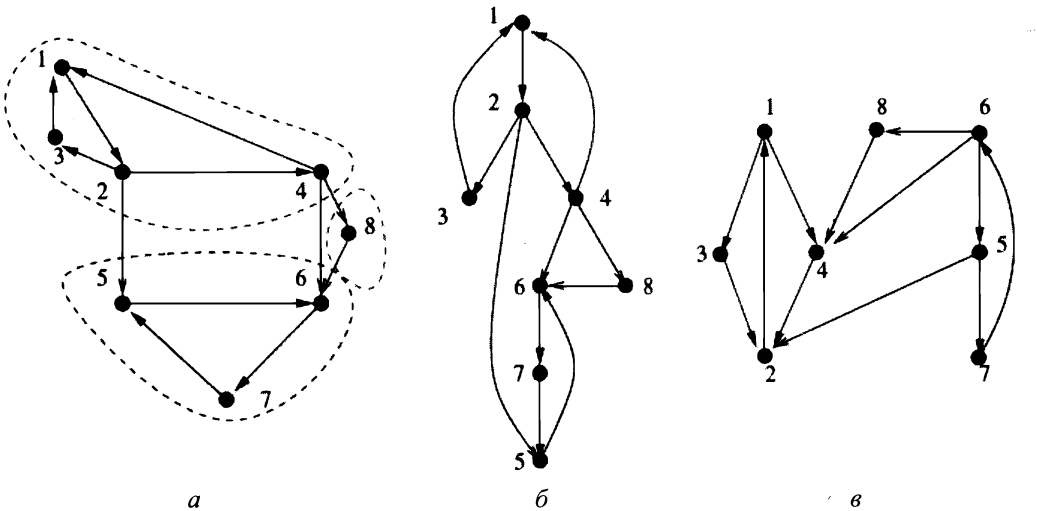



**Рис. 7.15.** Сильно связные компоненты графа (*а*) и соответствующее дерево обхода в глубину (*б*). Инвертируя конечный порядок вершин обхода в глубину с вершины 1, получим множество [3, 5, 7, 6, 8, 4, 2, 1], что определяет порядок вершин для второго обхода, на этот раз транспонированного графа $G^T$ (*в*)

**Листинг 7.25. Алгоритм разложения графа на сильно связные компоненты**

```
void strong_components(graph *g) {
    graph *gt;       /* Транспонирование графа g */
    int i;           /* Счетчик */
    int v;           /* Вершина в компоненте */

    init_stack(&dfslorder);
    initialize_search(g);
    for (i = 1; i <= (g->nvertices); i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }

    gt = transpose(g);
    initialize_search(gt);
```

```
    components_found = 0;
    while (!empty_stack(&dfslorder)) {
        v = pop(&dfslorder);
        if (!discovered[v]) {
            components_found ++;
            printf("Component %d:", components_found);
            dfs2(gt, v);
            printf("\n");
        }
    }
}
```

В первом обходе вершины помещаются в стек в порядке, обратном порядку их обработки, точно так же, как и в топологической сортировке (см. *разд. 7.10.1*). Такая связь имеет смысл — в бесконтурных ориентированных графах каждая вершина формирует свой собственный сильно связный компонент. Для бесконтурного ориентированного графа в самом верху стека находится вершина, из которой не достижимы никакие другие вершины. Учет ресурсов ведется точно так же, как и при топологической сортировке:

```
void process_vertex_late(int v) {
    push(&dfslorder,v);
}
```

Второй обход — на этот раз транспонированного графа — выполняется подобно алгоритму поиска компонентов связности (см. листинг 7.14), с тем исключением, что начальные вершины рассматриваются в том порядке, в котором они размещены в стеке. В каждом обходе с вершины $v$ открываются все вершины, достижимые из транспонированного графа $G^T$, т. е. вершины, из которых в графе $G$ достижима вершина $v$. Эти достижимые вершины определяют сильно связный компонент вершины $v$, т. к. они представляют минимально достижимые вершины в графе $G$:

```
void process_vertex_early2(int v) {
    printf(" %d", v);
}
```

Демонстрация правильности этого подхода довольно тонкая. Обратите внимание на то, что в первом обходе в глубину вершины помещаются в стек группами на основе их достижимости из последовательных начальных вершин в исходном ориентированном графе $G$. Таким образом, вершины верхней группы обладают тем свойством, что *ни одна из них* не достижима *ни из какой* вершины предыдущей группы. Второй обход, на этот раз транспонированного графа $G^T$, начинается с последней вершины $v$ графа $G$ и открывает все вершины в графе $G^T$, которые достижимы из вершины $v$ и из которых достижима вершина $v$, — иными словами те вершины, которые определяют сильно связный компонент графа.

# Замечания к главе

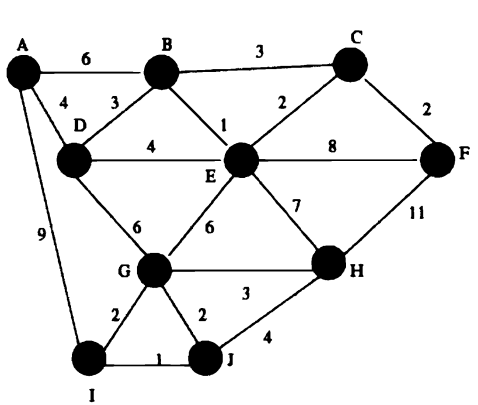
Рассмотренный в этой главе материал по обходу графов является расширенной версией материала из *главы 9* книги [SR03]. Самое лучшее описание библиотеки для работы с графами Combinatorica можно найти в старом [Ski90] и новом [PS03] издании книги,

посвященной работе с этой библиотекой. Доступное введение в теорию социальных сетей можно найти в книгах [Bar03], [EK10] и [Wat04]. Интерес к теории графов значительно возрос с появлением мультидисциплинарной области сетевых наук — см. книги [B+16] и [New18].

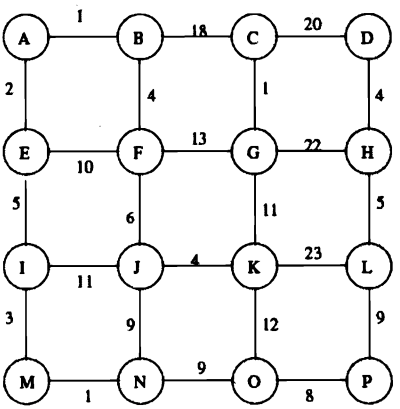
# 7.11. Упражнения

## Алгоритмы для эмуляции графов

1. *[3]* Даны взвешенные графы $G_1$ и $G_2$ (рис. 7.16).




Граф $G_1$




Граф $G_2$

**Рис. 7.16.** Примеры графов для упражнения 1

a) Укажите порядок вершин при обходе в ширину, начинающемся с вершины $A$. В случае конфликтов рассматривайте вершины в алфавитном порядке (т. е. $A$ предшествует $Z$).

b) Укажите порядок вершин при обходе в глубину, начинающемся с вершины $A$. В случае конфликтов рассматривайте вершины в алфавитном порядке (т. е. $A$ предшествует $Z$).

2. *[3]* Выполните топологическую сортировку на графе $G$, представленном на рис. 7.17.

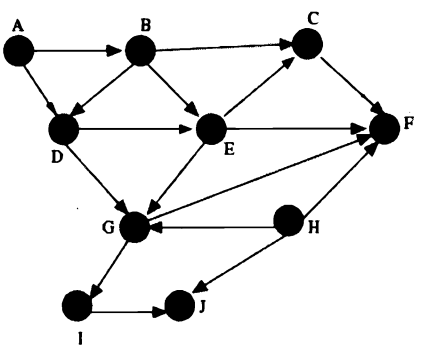



**Рис. 7.17.** Пример графа для упражнения 2

## Обход графов

3. *[3]* Докажите, что между любой парой вершин дерева существует однозначный путь.

4. *[3]* Докажите, что при обходе в ширину неориентированного графа $G$ каждое ребро может быть либо древесным, либо поперечным, где $x$ не является ни предшественником, ни потомком вершины $y$ в поперечном ребре $(x, y)$.

5. *[3]* Разработайте алгоритм с линейным временем исполнения для вычисления хроматического числа графов, где степень каждой вершины не выше второй. Хроматическое число любого двудольного графа равно 2. Должны такие графы быть двудольными?

6. *[3]* Имеется связный неориентированный граф $G$, содержащий $n$ вершин и $m$ ребер. Разработайте алгоритм с временем исполнения $O(n + m)$ для нахождения ребра, которое можно удалить из этого графа, не нарушая его связности, если такое ребро имеется. Можете ли вы сократить время исполнения решения до $O(n)$?

7. *[5]* В обходах в ширину и глубину неоткрытая вершина помечается как *открытая* при первом ее посещении и как *обработанная* после того, как она была полностью исследована. В любой момент времени в *открытом* состоянии могут находиться несколько вершин одновременно.

   a) Опишите граф из $n$ вершин с заданной вершиной $v$, у которого при обходе в ширину, начинающемся в вершине $v$, одновременно $\Theta(n)$ вершин находятся в *открытом* состоянии.

   b) Опишите граф из $n$ вершин с заданной вершиной $v$, у которого при обходе в глубину, начинающемся в вершине $v$, одновременно $\Theta(n)$ вершин находятся в *открытом* состоянии.

   c) Опишите граф из $n$ вершин с заданной вершиной $v$, у которого в некоторой точке обхода в глубину, начинающегося в вершине $v$, одновременно $\Theta(n)$ вершин находятся в *неоткрытом* состоянии, в то время как $\Theta(n)$ вершин находятся в *обработанном* состоянии. (Подсказка: следует иметь в виду, что при этом могут также существовать *открытые* вершины.)

8. *[4]* Возможно ли восстановить двоичное дерево (см. *разд. 3.4.1*), зная его обход в прямом порядке и симметричный обход? Если возможно, то опишите алгоритм для этого. Если невозможно, то предоставьте контрпример. Решите задачу, если даны обходы в прямом и обратном порядке.

9. *[3]* Предоставьте правильный эффективный алгоритм для преобразования одной структуры данных неориентированного графа $G$ в другую. Для каждого алгоритма предоставьте значение временной сложности, полагая, что граф состоит из $n$ вершин и $m$ ребер.

   a) Преобразовать матрицу смежности в списки смежности.

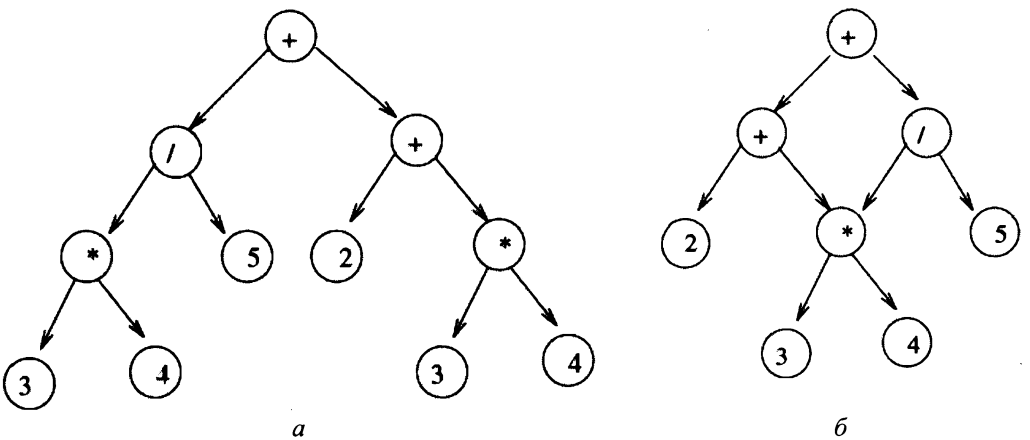
   b) Преобразовать список смежности в матрицу инцидентности. Матрица инцидентности $M$ содержит строку для каждой вершины и столбец для каждого ребра. Если вершина $i$ является частью ребра $j$, то $M[i, j] = 1$, в противном случае $M[i, j] = 0$.

   c) Преобразовать матрицу инцидентности в списки смежности.

10. *[3]* Дано арифметическое выражение в виде дерева. Каждый лист дерева является целым числом, а каждый внутренний узел представляет одну из стандартных арифметических операций (+, –, *, /). Для примера представление выражения: 2 + 3 * 4 + (3 * 4) / 5

в виде дерева показано на рис. 7.18, *а*. Разработайте алгоритм с временем исполнения $O(n)$ для вычисления такого выражения, где $n$ — количество узлов в дереве.

11. *[5]* Арифметическое выражение представлено в виде бесконтурного ориентированного графа, из которого удалены общие вложенные выражения. Каждый лист является целым числом, а каждый внутренний узел представляет одну из стандартных арифметических операций (+, –, *, /). Для примера представление выражения: 2 + 3 * 4 + (3 * 4) / 5 в виде бесконтурного ориентированного графа показано на рис. 7.18, *б*. Разработайте алгоритм для вычисления такого выражения за время $O(n + m)$, где $n$ — количество вершин бесконтурного ориентированного графа, а $m$ — количество ребер. (Подсказка: для достижения требуемой эффективности модифицируйте алгоритм для случая с деревом.)




*а* *б*

**Рис. 7.18.** Представление выражения 2 + 3 * 4 + (3 * 4) / 5 в виде дерева (*а*) и бесконтурного ориентированного графа (*б*)

12. *[8]* В *разд. 7.4* описывается алгоритм для эффективного создания двойственного графа триангуляции, который не гарантирует линейного времени исполнения. Для этой задачи разработайте алгоритм с линейным временем исполнения в наихудшем случае.

## Приложения

13. *[3]* Цель игры Chutes and Ladders (спуски и лестницы) состоит в перемещении по доске из $n$ клеток из клетки 1 в клетку $n$. Количество клеток, на сколько можно переместиться за один ход, определяется броском шестигранной игральной кости. Определенные клетки доски соединяются спусками и лестницами. Если конечная клетка хода игрока содержит начало спуска, то он перемещается на более низкую клетку в конце спуска. Аналогично, если конечная клетка хода игрока содержит начало лестницы, то он перемещается на более высокую клетку в ее конце. Предположим, что в вашем распоряжении мошенническая кость, и вы можете управлять результатом каждого броска. Разработайте алгоритм для определения минимального количества бросков, чтобы выиграть.

14. *[3]* Столбы «цветок сливы» (plum blossom poles) — тренировочное оборудование для кун-фу. Это $n$ больших столбов, закопанных в землю, при этом каждый столб $p_i$ находится в позиции $(x_i, y_i)$. Тренировка заключается в переходе от одного столба к другому по их верхушкам. Чтобы не потерять равновесие, каждый шаг должен быть больше, чем

$d$ метров, но меньше, чем $2d$ метров. Разработайте эффективный алгоритм, чтобы найти безопасный путь от столба $p_s$ к столбу $p_t$, если такой имеется.

15. *[5]* Вы планируете схему рассадки гостей на свадьбе по списку $V$. Для каждого гостя $g$ в вашем распоряжении есть список всех других гостей, которые с ним не ладят. Кстати, это чувство взаимное — т. е. если гость $h$ не ладит с гостем $g$, тогда и гость $g$ не ладит с гостем $h$. Ваша цель — рассадить гостей таким образом, чтобы за одним столом не было ни одной пары гостей, которые не ладят друг с другом. На свадьбе будут только два стола. Разработайте эффективный алгоритм, чтобы найти приемлемую схему рассадки гостей, если таковая существует.

## Разработка алгоритмов

16. [5] *Квадратом* (square) ориентированного графа $G = (V, E)$ называется граф $G^2 = (V, E^2)$ при условии, что $(u, w) \in E^2$ тогда и только тогда, когда существует такая вершина $v \in V$, для которой $(u, v) \in E$ и $(v, w) \in E$, — т. е. от вершины $u$ к вершине $w$ существует путь, состоящий ровно из двух ребер. Предоставьте эффективные алгоритмы для создания списков и матриц смежности для таких графов.

17. [5] *Вершинным покрытием* (vertex cover) графа $G = (V, E)$ называется такое подмножество вершин $V'$, в котором каждое ребро в $E$ инцидентно по крайней мере одной вершине в $V'$.

    a) Разработайте эффективный алгоритм поиска вершинного покрытия минимального размера, если $G$ является деревом.

    b) Пусть $G = (V, E)$ — это такой граф, в котором вес каждой вершины равен степени ее ребра. Разработайте эффективный алгоритм поиска вершинного покрытия графа минимального веса.

    c) Пусть $G = (V, E)$ — дерево с вершинами с произвольным весом. Разработайте эффективный алгоритм поиска вершинного покрытия графа минимального веса.

18. *[3] Вершинным покрытием* графа $G = (V, E)$ называется такое подмножество вершин $V'$, в котором каждое ребро в $E$ инцидентно по крайней мере одной вершине в $V'$. Если мы удалим все листья из любого дерева обхода в глубину графа $G$, то будут ли оставшиеся вершины составлять вершинное покрытие графа $G$? Если да, предоставьте доказательство, если нет, приведите контрпример.

19. *[5] Независимым множеством* (independent set) неориентированного графа $G = (V, E)$ называется такое подмножество вершин $U$, для которого $E$ не содержит ни одного ребра, обе вершины которого являлись бы членами $U$.

    a) Разработайте эффективный алгоритм поиска независимого множества максимального размера, если $G$ является деревом.

    b) Пусть $G = (V, E)$ — дерево, каждая вершина которого имеет вес, равный степени этой вершины. Разработайте эффективный алгоритм поиска независимого множества максимального веса дерева $G$.

    c) Пусть $G = (V, E)$ — дерево с вершинами с произвольно присвоенным весом. Разработайте эффективный алгоритм поиска независимого множества максимального веса дерева $G$.

20. *[5] Вершинным покрытием* (vertex cover) графа $G = (V, E)$ называется такое подмножество вершин $V'$, для которого каждое ребро в $E$ инцидентно по крайней мере одной

вершине в $V'$. *Независимым множеством* (independent set) графа $G = (V, E)$ называется такое подмножество вершин $V' \in V$, в котором $E$ не содержит ни одного ребра, обе вершины которого являлись бы членами $V'$. Разработайте эффективный алгоритм для проверки, содержит ли граф $G$ независимое вершинное покрытие. К какой классической задаче на графах сводится эта задача?

21. *[5]* Нужно определить, содержит ли заданный неориентированный граф $G = (V, E)$ *треугольник*, т. е. цикл длиной 3.

    a) Разработайте алгоритм с временем исполнения $O(|V|^3)$ поиска треугольника в графе, если таковой имеется.

    b) Усовершенствуйте ваш алгоритм для исполнения за время $O(|V| \cdot |E|)$. Можно полагать, что $|V| \leq |E|$.

    Обратите внимание на то, что эти пределы позволяют оценить время для преобразования представления графа $G$ из матрицы смежности в список смежности (или в противоположном направлении).

22. *[5]* Имеется набор фильмов $M_1$, $M_2$, ..., $M_k$ и группа клиентов, причем каждый из них указывает два фильма, которые он бы хотел посмотреть в ближайшие выходные. Фильмы показываются по вечерам в субботу и воскресенье. Одновременно могут демонстрироваться несколько фильмов.

    Вам нужно решить, какие фильмы следует показывать в субботу, а какие — в воскресенье, таким образом, чтобы каждый клиент смог посмотреть указанные им фильмы. Возможно ли составить расписание, согласно которому каждый фильм показывается только один раз? Разработайте эффективный алгоритм поиска такого расписания.

23. *[5]* Диаметр дерева $T = (V, E)$ задается формулой

$$\max_{u,v \in V} \delta(u, v),$$

    где $\delta(u, v)$ — количество ребер в пути от вершины $u$ к вершине $v$. Опишите эффективный алгоритм для вычисления диаметра дерева, докажите его правильность и проанализируйте время его исполнения.

24. *[5]* Имеется неориентированный граф $G$ из $n$ вершин и $m$ ребер и целое число $k$. Разработайте алгоритм с временем исполнения $O(m + n)$ для поиска максимального порожденного подграфа $H$ графа $G$, каждая вершина которого имеет степень $\geq k$, или докажите, что такого графа не существует. Граф $F = (U, R)$ называется *порожденным подграфом* (induced subgraph) графа $G = (V, E)$, если множество его вершин $U$ является подмножеством множества вершин $V$ графа $G$, а $R$ содержит все ребра графа $G$, обе вершины которых являются членами $U$.

25. *[6]* Даны две вершины $v$ и $w$ в невзвешенном ориентированном графе $G = (V, E)$. Разработайте алгоритм с линейным временем исполнения для поиска разных кратчайших путей (не обязательно не имеющих общих вершин) между $v$ и $w$.

26. *[6]* Разработайте алгоритм с линейным временем исполнения для удаления всех вершин второй степени из графа путем замены ребер $(u, v)$ и $(v, w)$ ребром $(u, w)$. Алгоритм также должен удалить множественные копии ребер, оставив только одно ребро. Обратите внимание, что удаление копий ребра может создать новую вершину второй степени, которую нужно будет удалить, а удаление вершины второй степени может создать кратные ребра, которые также нужно будет удалить.

## Ориентированные графы

27. [3] *Обратным* ориентированным графом $G = (V, E)$ называется другой ориентированный граф $G^R = (V, E^R)$ с таким же множеством вершин, но все ребра которого инвертированы, — т. е. $E^R = \{(v, u) : (u, v) \in E\}$. Разработайте алгоритм с временем исполнения $O(n + m)$ для вычисления обратного графа в формате списка смежностей для графа с $n$ вершинами и $m$ ребрами.

28. *[5]* Перед вами стоит задача выстроить $n$ непослушных детей друг за другом, причем у вас имеется список из $m$ утверждений типа «$i$ ненавидит $j$». Если $i$ ненавидит $j$, то будет разумно не ставить $i$ позади $j$, т. к. $i$ может что-нибудь швырнуть в $j$.

    a) Разработайте алгоритм с временем исполнения $O(m + n)$ для выстраивания детей с соблюдением имеющихся в списке условий. Если такое упорядочивание невозможно, алгоритм должен сообщить об этом.

    b) Допустим, что вам нужно выстроить детей в несколько рядов таким образом, что если $i$ ненавидит $j$, то $i$ должен стоять в каком-нибудь ряду впереди $j$. Разработайте алгоритм для определения минимального количества рядов, если это возможно.

29. *[3]* Определенная академическая программа состоит из $n$ обязательных курсов, некоторые пары которых связаны предварительными условиями, согласно которым $(x, y)$ означает, что прежде чем взять курс $y$, нужно пройти курс $x$. Как бы вы подошли к анализу пар курсов с предварительными условиями, чтобы обеспечить гарантированное завершение программы?

30. *[5]* Для карточной игры-пасьянса Gotcha (Ага, попался) используется колода, содержащая $n$ различных карт (лицом вверх) и $m$ пар карт Gotcha $(i, j)$, в которых карта $i$ должна играться перед картой $j$. Карты последовательно выбираются из колоды, и выигрышем считается достижение конца колоды без каких-либо нарушений ограничений Gotcha. Разработайте эффективный алгоритм для нахождения выигрышного порядка выбора карт, если такой существует.

31. *[5]* Имеется список из $n$ слов длиной $k$ символов каждое на неизвестном вам языке, но вы знаете, что все слова в списке упорядочены в алфавитном порядке. Воссоздайте порядок букв (символов) $\alpha$ в алфавите этого языка.

    Например, для строк *{QQZ, QZZ, XQZ, XQX, XXX}* порядок символов должен быть Q, Z, X.

    a) Разработайте эффективный алгоритм для воссоздания такого порядка символов. (Подсказка: используйте граф, в котором каждый узел представляет одну букву.)

    b) Каким будет время исполнения этого алгоритма в зависимости от величин $n$, $k$ и $\alpha$?

32. *[3] Слабо связным компонентом* ориентированного графа называется связный компонент, в котором игнорируются направления ребер. Определите максимальное количество компонентов, на которое можно уменьшить количество слабо связных компонентов ориентированного графа, добавив в него одно ориентированное ребро. Что можно сказать по поводу количества сильно связных компонентов?

33. *[5]* Разработайте алгоритм с линейным временем исполнения, который для неориентированного графа $G$ и отдельного ребра $e$ в нем определяет, есть ли в графе $G$ цикл, содержащий это ребро $e$.

34. *[5] Ориентированным деревом* (arborescence) ориентированного графа $G$ называется такое корневое дерево, что существует ориентированный путь от корня ко всем другим

вершинам графа. Разработайте правильный эффективный алгоритм для тестирования, содержит ли граф $G$ ориентированное дерево. Укажите временную сложность алгоритма.

35. *[5] Истоком* орграфа $G = (V, E)$ называется такая вершина $v$, из которой все другие вершины графа $G$ достижимы ориентированным путем.

36. a) Разработайте алгоритм с временем исполнения $O(n + m)$ (где $n = |V|$ и $m = |E|$) для проверки того, является ли заданная вершина $v$ истоком для графа $G$.

37. b) Разработайте алгоритм с временем исполнения $O(n + m)$ (где $n = |V|$ и $m = |E|$) для проверки того, содержит ли граф $G$ исток.

38. *[8]* Имеется ориентированный граф $G$. Граф $G$ называется *k-циклическим* (с цикломатическим числом $k$), если каждый (не обязательно простой) цикл графа содержит самое большее $k$ различных узлов. Разработайте алгоритм с линейным временем исполнения и вводом $G$ и $k$ для определения, является ли граф G $k$-циклическим. Предоставьте доказательство правильности своего алгоритма и его времени исполнения.

39. *[9] Турниром* (tournament) называется полный ориентированный граф — т. е. такой граф $G = (V, E)$, в котором для всех $u, v \in V$ только или $(u, v)$, или $(v, u)$ принадлежит множеству $E$. Покажите, что каждый турнир содержит гамильтонов путь, т. е. путь, который проходит через каждую вершину только один раз. Разработайте алгоритм поиска этого пути.

## Шарниры графа

40. *[5] Шарниром* связного графа $G$ называется вершина, удаление которой разъединяет граф. Дан граф $G$ из $n$ вершин и $m$ ребер. Разработайте алгоритм с временем исполнения $O(n + m)$ для поиска вершины в графе $G$, *не являющейся* шарниром, — т. е. вершины, чье удаление не разъединяет граф.

41. *[5]* По условиям предыдущей задачи разработайте алгоритм с временем исполнения $O(n + m)$ для поиска такой последовательности удалений $n$ вершин, в которой ни одно удаление не разъединяет граф. (Подсказка: рассмотрите возможность обхода в глубину и в ширину.)

42. *[3]* Дан связный неориентированный граф $G$. Ребро $e$, удаление которого разъединяет граф, называется *мостом*. Должен ли каждый мост $e$ быть ребром в дереве обхода в глубину графа $G$? Если да, предоставьте доказательство, если нет, приведите контрпример.

43. *[5]* В городе, где все улицы с двусторонним движением, решили изменить их все на одностороннее движение. При этом требуется обеспечить сильную связность полученной сети, чтобы можно было проехать в любую точку города и назад, не нарушая при этом правила дорожного движения.

    a) Пусть $G$ — неориентированный граф исходной дорожной сети. Докажите, что существует способ правильно сориентировать/направить ребра графа $G$ при условии, что он не содержит моста.

    b) Разработайте эффективный алгоритм для ориентирования ребер безмостового графа $G$, чтобы получить сильно связный граф.

## Задачи, предлагаемые на собеседовании

44. *[5]* Какие структуры данных используются при обходе в глубину и при обходе в ширину?
45. *[4]* Напишите функцию, которая обходит дерево двоичного поиска и возвращает *i*-й по порядку узел.

## LeetCode

1. **https://leetcode.com/problems/minimum-height-trees/**
2. **https://leetcode.com/problems/redundant-connection/**
3. **https://leetcode.com/problems/course-schedule/**

## HackerRank

1. **https://www.hackerrank.com/challenges/bfsshortreach/**
2. **https://www.hackerrank.com/challenges/dfs-edges/**
3. **https://www.hackerrank.com/challenges/even-tree/**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org**:

«Bicoloring», глава 9, задача 10004.

1. «Playing with Wheels», глава 9, задача 10067.
2. «The Tourist Guide», глава 9, задача 10099.
3. «Edit Step Ladders», глава 9, задача 10029.
4. «Tower of Cubes», глава 9, задача 10051.

# Алгоритмы для работы со взвешенными графами

Рассмотренные в *главе 7* структуры данных и алгоритмы обхода предоставляют базовые конструктивные блоки для выполнения любых вычислений на графах. Но все эти алгоритмы применялись для *невзвешенных графов* (unweighted graphs) — т. е. графов, в которых все ребра имеют одинаковый вес.

Но для *взвешенных графов* (weighted graph) существует отдельная область задач. В частности, ребрам графов дорожных сетей присваиваются такие числовые значения, как длина, скорость движения или пропускная способность того или иного отрезка дороги. Определение кратчайшего пути в таких графах оказывается более сложной задачей, чем обход в ширину в невзвешенных графах, но открывает путь к обширному диапазону приложений.

Структуры данных, с которыми мы познакомились в *главе 7*, поддерживают графы со взвешенными ребрами неявно, но сейчас мы сделаем эту возможность явной. Для этого мы опять сначала определим структуру списка смежности, состоящую из массива связных списков, в которых ребра, исходящие из вершины $x$, указываются в списке `edges[x]` (листинг 8.1).

**Листинг 8.1. Определение структуры списка смежности**

```
typedef struct {
   edgenode *edges[MAXV+1];    /* Информация о смежности */
   int degree[MAXV+1];         /* Степень каждой вершины */
   int nvertices;              /* Количество вершин в графе */
   int nedges;                 /* Количество ребер в графе */
   int directed;               /* Граф ориентированный? */
} graph;
```

Каждая переменная `edgenode` представляет собой запись из трех полей (листинг 8.2). Первое поле описывает вторую конечную точку ребра (`y`), второе (`weight`) предоставляет возможность присваивать ребру вес, а третье (`next`) указывает на следующее ребро в списке.

**Листинг 8.2. Структура переменной `edgenode`**

```
typedef struct edgenode {
   int y;                      /* Информация о смежности */
   int weight;                 /* Вес ребра, если есть */
   struct edgenode *next;      /* Следующее ребро в списке */
} edgenode;
```

Далее мы рассмотрим несколько сложных алгоритмов для работы со взвешенными графами, использующих эту структуру данных, включая алгоритмы поиска минимальных остовных деревьев, кратчайших маршрутов и максимальных потоков. То обстоятельство, что для всех этих задач оптимизации существуют эффективные решения, привлекает к ним наше особое внимание. Вспомните, что для первой рассмотренной нами задачи для взвешенных графов (задачи коммивояжера) детерминистического алгоритма решения не существует.

# 8.1. Минимальные остовные деревья

*Остовным деревом* (spanning tree) связного графа $G = (V, E)$ называется подмножество ребер множества $E$, которые создают дерево, содержащее все вершины $V$. В случае взвешенных графов особый интерес представляют минимальные остовные деревья — т. е. остовные деревья с минимальной суммой весов ребер.

Минимальные остовные деревья позволяют решить задачу, в которой требуется соединить множество точек (представляющих, например, города, дома, перекрестки и другие объекты) наименьшим объемом дорожного полотна, проводов, труб и т. п. *Любое* дерево — это, в сущности, наименьший (по количеству ребер) возможный связный граф, но минимальное остовное дерево является наименьшим связным графом по весу ребер. В геометрических задачах набор точек $p_1, ..., p_n$ определяет полный граф, в котором ребру $(v_i, v_j)$ присваивается вес, равный расстоянию между точками $p_i$ и $p_j$. На рис. 8.1 показан пример геометрического минимального остовного дерева. Дополнительная информация о минимальных остовных деревьях приводится в *разд 18.3*.

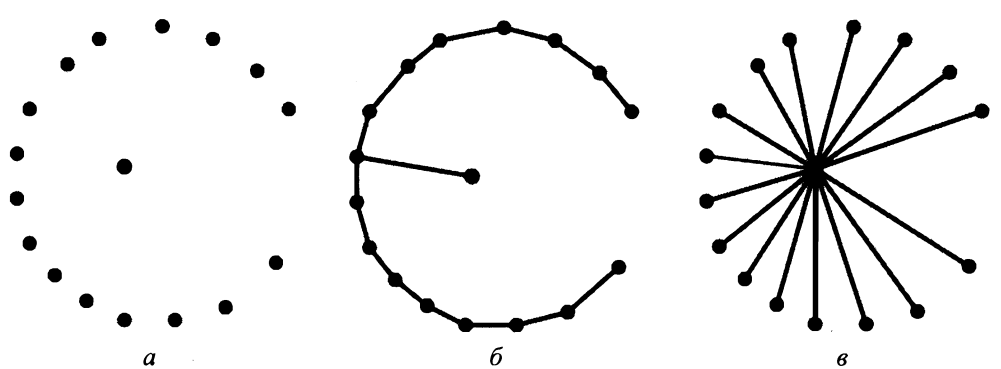



**Рис. 8.1.** Полный взвешенный граф, определяемый расстоянием между набором точек (*а*), его минимальное остовное дерево (*б*) и кратчайший маршрут от центра дерева (*в*)

Минимальное остовное дерево имеет наименьший общий вес ребер изо всех возможных остовных деревьев. Но граф может содержать несколько минимальных остовных деревьев. Более того, все остовные деревья невзвешенного графа (или графа со всеми ребрами одинакового веса) являются минимальными остовными деревьями, поскольку каждое из них содержит ровно $n - 1$ ребер одинакового веса. Такое остовное дерево можно найти с помощью алгоритма обхода в глубину или ширину. Найти минимальное остовное дерево во взвешенном графе гораздо труднее. Но в последующих разделах

рассматриваются два разных алгоритма для решения этой задачи, каждый из которых демонстрирует пригодность определенных эвристических «жадных» алгоритмов.

### 8.1.1. Алгоритм Прима

Алгоритм Прима для построения минимального остовного дерева начинает обход с одной вершины и создает дерево, добавляя по одному ребру, пока не будут включены все вершины.

«Жадные» алгоритмы принимают решение по следующему шагу, выбирая наилучшее локальное решение изо всех возможных вариантов, не принимая во внимание глобальную структуру. Так как мы ищем дерево с минимальным весом, то естественно предложить «жадный» алгоритм, последовательно выбирающий ребра с наименьшим весом, которые увеличат количество вершин дерева. Псевдокод алгоритма для построения минимального остовного дерева (Minimum Spanning Tree, (MST) приведен в листинге 8.3.

**Листинг 8.3. Алгоритм Прима**

```
Prim-MST(G)
  Выбираем произвольную вершину s, с которой надо начинать
                                    построение дерева T_prim
    While (остаются вершины, не включенные в дерево)
     Находим ребро минимального веса между деревом и вершиной вне дерева
     Добавляем выбранное ребро и вершину в дерево T_prim
```

Очевидно, что алгоритм Прима создает остовное дерево, т. к. добавление ребра между вершиной в дереве и вершиной вне дерева не может создать цикл. Но почему именно это остовное дерево должно иметь наименьший вес изо *всех* остовных деревьев? Мы видели достаточно примеров других «жадных» эвристических алгоритмов, которые не дают оптимального общего решения. Поэтому нужно быть особенно осторожным, чтобы доказать любое такое утверждение.

В нашем случае мы применим метод доказательства от противного. Допустим, что существует граф $G$, для которого алгоритм Прима не возвращает минимальное остовное дерево. Так как мы создаем дерево инкрементным способом, то это означает, что существует момент, в который было принято неправильное решение. Перед добавлением ребра $(x, y)$ дерево $T_{\text{prim}}$ состояло из набора ребер, являющихся поддеревом определенного минимального остовного дерева $T_{\text{min}}$, но остовное дерево, полученное добавлением к этому поддереву ребра $(x, y)$, больше не является каким-либо возможным минимальным остовным деревом (рис. 8.2, *а*).

Но как это могло случиться? На рис. 8.2, *б* мы видим, что в остовном дереве $T_{\text{min}}$ должен быть путь $p$ от вершины $x$ к вершине $y$. Этот путь должен содержать ребро $(v_1, v_2)$, в котором вершина $v_1$ уже находится в дереве $T_{\text{prim}}$, а вершина $v_2$ — нет. Вес этого ребра должен быть по крайней мере равен весу ребра $(x, y)$, иначе алгоритм Прима выбрал бы это ребро до ребра $(x, y)$, когда у него была такая возможность. Удалив из дерева $T_{\text{min}}$ ребро $(v_1, v_2)$ и вставив вместо него ребро $(x, y)$, мы получим остовное дерево с весом не большим, чем прежнее дерево. Это означает, что выбор алгоритмом

Прима ребра $(x, y)$ не мог быть ошибочным. Таким образом, доказательство от противного показывает, что создаваемое алгоритмом Прима остовное дерево должно быть минимальным.

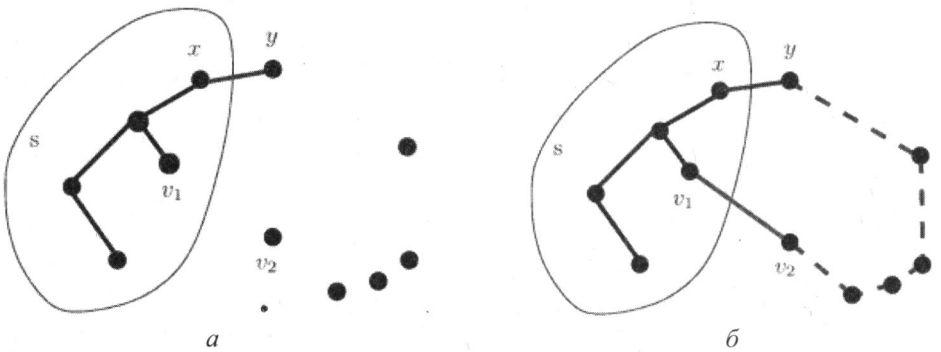



**Рис. 8.2.** Выдает ли алгоритм Прима неправильное решение? Нет, поскольку выбор ребра $(x,y)$ перед ребром $(v_1,v_2)$ подразумевает, что *вес* $(v_1, v_2) \geq$ *вес* $(x, y)$

### Реализация

Алгоритм Прима создает минимальное остовное дерево поэтапно, начиная с указанной вершины. На каждом этапе мы вставляем в остовное дерево одну новую вершину. «Жадного» алгоритма достаточно для гарантии правильности выбора — чтобы соединить вершины в дереве с вершиной вне дерева, он всегда выбирает ребро с наименьшим весом. Самой простой реализацией этого алгоритма было бы представлять каждую вершину булевой переменной, указывающей, находится ли уже эта вершина в дереве (массив `intree` в листинге 8.4), а потом просмотреть все ребра на каждом этапе, чтобы найти ребро с минимальным весом и единственной вершиной `intree`. В листинге 8.4 приведена реализация алгоритма Прима.

**Листинг 8.4. Реализация алгоритма Прима**

```
int prim(graph *g, int start) {
    int i;                  /* Счетчик */
    edgenode *p;            /* Временный указатель */
    bool intree[MAXV+1];    /* Вершина уже в дереве? */
    int distance[MAXV+1];   /* Стоимость добавления в дерево */
    int v;                  /* Текущая вершина для обработки */
    int w;                  /* Кандидат на следующую вершину */
    int dist;               /* Наименьшая стоимость расширения дерева */
    int weight = 0;         /* Вес дерева */

    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;

    }
```

```
    distance[start] = 0;
    v = start;

    while (!intree[v]) {
        intree[v] = true;
        if (v != start) {
            printf("edge (%d,%d) in tree \n",parent[v],v);
            weight = weight + dist;
        }
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            if ((distance[w] > p->weight) && (!intree[w])) {
                distance[w] = p->weight;
                parent[w] = v;
            }
            p = p->next;
        }

        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++) {
        if ((!intree[i]) && (dist > distance[i])) {
            dist = distance[i];
                v = i;
            }
        }
    }

    return(weight);
}
```

В приведенной процедуре поддерживается информация о весе каждого ребра, связывающего с деревом каждую находящуюся вне дерева вершину. Из множества этих ребер в каждой следующей итерации цикла к дереву добавляется ребро с наименьшим весом. После каждой вставки ребра необходимо обновлять стоимость (в виде весов ребер) просмотра вершин, не входящих в дерево. Но т. к. единственным изменением в дереве является последняя добавленная вершина, то все возможные обновления весов будут определяться исходящими ребрами этой вершины.

### Анализ

Итак, алгоритм Прима является правильным, но насколько он эффективен? Это будет зависеть от используемых для его реализации структур данных. В псевдокоде алгоритм Прима исполняет $n$ циклов, просматривая все $m$ ребер в каждом цикле, что дает нам алгоритм с временной сложностью, равной $O(mn)$.

Но в нашей реализации проверка всех ребер в каждом цикле не выполняется. Мы рассматриваем не более $n$ известных ребер с наименьшим весом, представленных в массиве `parent`, и не более $n$ ребер, исходящих из последней добавленной вершины $v$ для обновления этого массива. Благодаря наличию булева флага для каждой вершины, ука-

зывающего, находится ли эта вершина в дереве, мы можем за постоянное время проверить, связывает ли текущее ребро дерево с вершиной вне дерева.

Общее время исполнения алгоритма Прима равно $O(n^2)$, что является хорошей иллюстрацией того, как выбор удачной структуры данных способствует ускорению работы алгоритмов. Более того, применение более сложной структуры данных в виде очереди с приоритетами позволяет осуществить реализацию алгоритма Прима с временной сложностью, равной $O(m + n \lg n)$, поскольку удается быстрее найти ребро с минимальным весом для расширения дерева в каждом цикле.

Само минимальное остовное дерево можно реконструировать двумя разными способами. Самым простым способом было бы дополнить код в листинге 8.4 операторами для вывода обнаруженных ребер и общего веса всех выбранных ребер, чтобы получить общую стоимость. В качестве альтернативы можно закодировать топологию дерева в массиве `parent`, чтобы она совместно с первоначальным графом предоставляла всю информацию о ребрах в минимальном остовном дереве.

## 8.1.2. Алгоритм Крускала

Алгоритм Крускала является альтернативным подходом к построению минимальных остовных деревьев, который оказывается более эффективным на разреженных графах. Подобно алгоритму Прима, алгоритм Крускала является «жадным», но, в отличие от него, он не создает дерево, начиная с определенной вершины. Как показано на рис. 8.3, алгоритм Крускала может предложить иное остовное дерево, чем алгоритм Прима, хотя оба эти дерева будут иметь одинаковый вес.

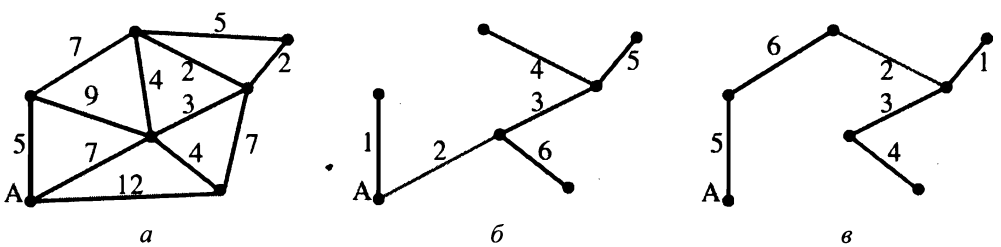



**Рис. 8.3.** Граф $G$ (*а*) и минимальные остовные деревья, создаваемые алгоритмами Прима (*б*) и Крускала (*в*). Цифры у ребер указывают порядок их вставки, приоритет вставки равнозначных ребер определяется произвольным путем

Алгоритм Крускала наращивает связные компоненты вершин, создавая в конечном счете полное минимальное остовное дерево. Первоначально каждая вершина является отдельным компонентом будущего дерева. Алгоритм последовательно ищет ребро для добавления в расширяющийся лес путем поиска самого легкого ребра среди всех ребер, соединяющих два дерева в лесу. При этом выполняется проверка, не находятся ли обе конечные точки ребра-кандидата в одном и том же связном компоненте. В случае положительного результата проверки такое ребро отбрасывается, поскольку добавление его создало бы цикл. Если же конечные точки находятся в разных компонентах, то ребро принимается и соединяет два компонента в один. Так как каждый связный ком-

понент всегда является деревом, то выполнять явную проверку на циклы никогда нет необходимости.

В листинге 8.5 приводится псевдокод алгоритма Крускала.

**Листинг 8.5. Псевдокод алгоритма Крускала**

```
Kruskal-MST(G)
    Put the edges into a priority queue ordered by increasing weight.
    Помещаем ребра в очередь с приоритетами, упорядоченные по возрастанию по весу
    count = 0
    while (count < n - 1) do
        рассматриваем следующее ребро (v,w)
        if (component (v) _= component(w))
            increment count
            добавляем (v,w) в дерево T_kruskal
            объединяем component(v) и component(w)
```

Так как этот алгоритм вставляет $n - 1$ ребер, не порождая циклы, он должен создавать остовное дерево для любого связного графа. Но почему это остовное дерево должно быть *минимальным*? Будем доказывать от противного. Допустим, что оно не является таковым. Так же, как и в доказательстве правильности алгоритма Прима, допустим, что существует какой-то определенный граф $G$, для которого он возвращает неправильный ответ.

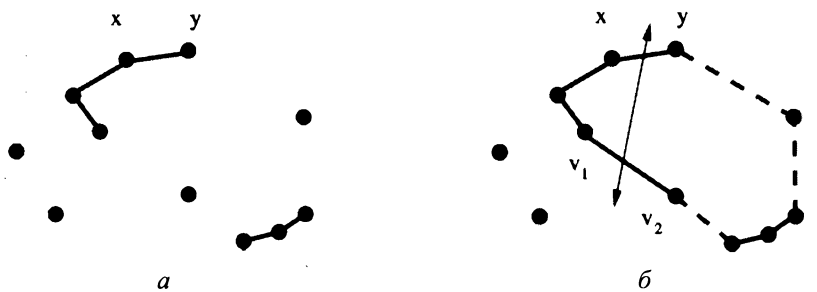



**Рис. 8.4.** Мог бы алгоритм Крускала выдать неправильное решение, выбрав ребро $(x, y)$ (*а*)? Нет, не мог бы, потому что вставляемое позже ребро $(v_1, v_2)$ (*б*) должно быть тяжелее, чем ребро $(x, y)$

В частности, в графе $G$ должно существовать ребро $(x, y)$, первоначальная вставка которого в дерево $T_{kruskal}$ не дает ему быть минимальным остовным деревом $T_{\min}$. Вставка такого ребра $(x, y)$ в дерево $T_{\min}$ создаст цикл с путем от вершины $x$ к вершине $y$ (рис. 8.4, *а*). Так как вершины $x$ и $y$ были в разных компонентах во время вставки ребра $(x, y)$, то по крайней мере одно ребро — скажем, ребро $(v_1, v_2)$, — в этом пути (рис. 8.4, *б*) должно было бы быть проверено алгоритмом Крускала позже, чем ребро $(x, y)$. Но это означает, что вес $weight(v_1, v_2) \geq weight(x, y)$, поэтому обмен местами этих двух ребер дает нам дерево с весом самое большее $T_{\min}$. Поэтому выбор ребра $(x, y)$ не мог быть ошибочным, из чего следует правильность алгоритма.

Какова временная сложность алгоритма Крускала? Упорядочивание $m$-го количества ребер занимает время $O(m \lg m)$. Цикл `while` выполняет самое большее $m$ итераций, в каждой из которых проверяется связь двух деревьев и ребро. При наиболее очевидном подходе это можно реализовать посредством поиска в ширину или глубину в разреженном подграфе, имеющем самое большее $m$ ребер и $n$ вершин, что дает нам алгоритм с временем исполнения $O(mn)$.

Но если удастся выполнять проверку компонентов быстрее, чем за время $O(n)$, то можно получить более эффективную реализацию алгоритма. В действительности, рассматриваемая в *разд. 8.1.3* сложная структура данных непересекающихся множеств (union-find или disjoint sets) позволяет выполнять такую проверку за время $O(\lg n)$. С использованием этой структуры алгоритм Крускала исполняется за время $O(m \lg m)$, что быстрее, чем время исполнения алгоритма Прима для разреженных графов. В очередной раз обращаю ваше внимание на эффект, оказываемый правильной структурой данных на реализацию простых алгоритмов.

### Реализация

Реализация алгоритма Крускала показана в листинге 8.6.

**Листинг 8.6. Реализация алгоритма Крускала**

```
int kruskal(graph *g) {
    int i;                  /* Счетчик */
    union_find s;           /* Структура данных union-find*/
    edge_pair e[MAXV+1];    /* Структура данных массива ребер */
    int weight=0;           /* Вес минимального остовного дерева */

    union_find_init(&s, g->nvertices);

    to_edge_array(g, e);      /* Сортируем ребра по возрастающему весу */
    qsort(&e,g->nedges, sizeof(edge_pair), &weight_compare);

    for (i = 0; i < (g->nedges); i++) {
        if (!same_component(&s, e[i].x, e[i].y)) {
            printf("edge (%d,%d) in MST\n", e[i].x, e[i].y);
            weight = weight + e[i].weight;
            union_sets(&s, e[i].x, e[i].y);
        }
    }

    return(weight);
}
```

## 8.1.3. Структура данных непересекающихся множеств

*Разбиением множества* (set partition) называется разбиение элементов некоего универсального множества (например, целых чисел от 1 до $n$) на несколько непересекающихся (disjointed) подмножеств таким образом, чтобы каждый элемент исходного множества находился ровно в одном из получившихся подмножеств. Разбиения множества

возникают естественным образом в таких задачах на графах, как связные компоненты (каждая вершина находится ровно в одном компоненте связности) и раскраска графа (в двудольном графе вершина может быть белого или черного цвета, но мы исключаем варианты отсутствия цвета или раскраски обоими цветами). Алгоритмы для генерирования разбиений множеств и связных объектов представлены в *разд. 17.6*.

Связные компоненты графа можно представить в виде разбиения множества. Для эффективного исполнения алгоритма Крускала нам нужна структура данных, обеспечивающая эффективную поддержку таких операций:

- *один компонент* ($v_1$, $v_2$) — выполняет проверку, находятся ли вершины $v_1$ и $v_2$ в одном и том же компоненте связности текущего графа;
- *объединить компоненты* ($C_1$, $C_2$) — объединяет указанную пару связных компонентов в один в ответ на вставку ребра между ними.

Каждая из двух очевидных структур-кандидатов для поддержки этих операций в действительности обеспечивает эффективную поддержку только одной из них. В частности, явно пометив каждый элемент номером его компонента, мы сможем выполнять проверку «один компонент» за постоянное время, но обновление номеров компонентов после слияния будет занимать линейное время. С другой стороны, операцию слияния компонентов можно рассматривать как вставку ребра в граф, но тогда нам нужно выполнить полный обход графа, чтобы найти связные компоненты в случае необходимости.

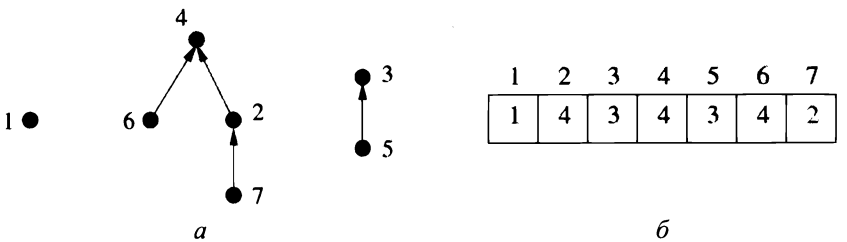



**Рис. 8.5.** Пример структуры данных «поиск-объединение»: в виде леса деревьев (*а*) и в виде массива указателей на родительские элементы (*б*)

Решением является применение специальной структуры данных непересекающихся множеств «поиск-объединение» (union-find), где каждое подмножество представляется в виде «обратного» дерева с указателями от узлов к их родителям (рис. 8.5). Каждый узел дерева содержит элемент множества, а *имя* множеству присваивается по корневому ключу. По причинам, которые станут понятными далее, мы для каждой вершины $v$ отслеживаем также информацию о количестве элементов поддерева, имеющего корень в этой вершине.

В листинге 8.7 приводится определение структуры данных непересекающихся множеств `union_find`.

**Листинг 8.7. Определение структуры данных `union_find`**

```
typedef struct {
   int p[SET_SIZE+1];      /* Родительский элемент */
```

```
    int size [SET_SIZE+1]; /* Количество элементов в поддереве i */
    int n;                 /* Количество элементов в множестве */
} union_find;
```

Реализуем необходимые нам операции над компонентами посредством двух более простых операций: `find` и `union`:

- `find(i)` — находит корень дерева, содержащего элемент *i*, переходя по указателям на родителей до тех пор, пока это возможно. Возвращает метку корня;
- `union(i,j)` — связывает корень одного из деревьев (скажем, дерева, содержащего элемент *i*) с корнем дерева, содержащего другой элемент (скажем, элемент *j*). Таким образом, операция `find(i)` теперь эквивалентна операции `find(j)`.

Мы стремимся минимизировать время исполнения *наихудшей возможной* последовательности операций `union` и `find`. Так как древесные структуры могут быть сильно разбалансированными, то нам необходимо ограничить высоту наших деревьев. Самым очевидным способом контроля высоты будет принятие правильного решения, какой из двух корней компонентов должен стать корнем объединенного компонента после каждого выполнения операции `union`.

Минимизировать высоту дерева можно, сделав меньшее дерево поддеревом большего. Почему это так? Потому что высота всех узлов в корневом поддереве остается одинаковой, но высота всех узлов, вставленных в это дерево, увеличивается на единицу. Таким образом, объединение с меньшим деревом позволяет оставить без изменений высоту большего дерева.

### Реализация

Реализация операций `union` и `find` приводится в листинге 8.8.

**Листинг 8.8. Реализация операций `union` и `find`**

```
void union_find_init(union_find *s, int n) {
    int i;                              /* Счетчик */

    for (i = 1; i <= n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }
    s->n = n;
}

int find(union_find *s, int x) {
    if (s->p[x] == x) {
        return(x);
    }
    return(find(s, s->p[x]));
}

void union_sets(union_find *s, int s1, int s2) {
    int r1, r2;          /* Корни подмножеств */
```

```
    r1 = find(s, s1);
    r2 = find(s, s2);

    if (r1 == r2) {
        return;            /* Уже находится в этом подмножестве */
    }

    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
        s->p[r2] = r1;
    } else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[r1] = r2;
    }
}

bool same_component(union_find *s, int s1, int s2) {
    return (find(s, s1) == find(s, s2));
}
```

**Анализ**

При каждом исполнении операции `union` дерево с меньшим количеством узлов становится потомком. Но каким высоким может быть такое дерево в зависимости от количества узлов в нем? Рассмотрим, каким может быть наименьшее возможное дерево с высотой $h$. Высота деревьев с одним узлом равна 1. Наименьшее дерево высотой 2 имеет два узла, полученные в результате объединения двух одноузловых деревьев. При вставке в объединение дополнительных одноузловых деревьев высота больше не увеличивается, поскольку эти деревья просто становятся потомками корневого дерева высотой 2. И только при объединении двух деревьев высотой 2 можно получить дерево высотой 3, имеющее как минимум четыре узла.

Уловили закономерность? Высота дерева увеличивается на единицу при удваивании количества узлов. Сколько раз мы можем удваивать количество узлов, пока не исчерпаем все $n$ узлов? Нам удастся сделать максимум $\lg n$ удваиваний. Таким образом, мы можем выполнять как операцию объединения `union`, так и операцию поиска `find` за время $O(\log n)$, что достаточно хорошо для эффективного использования алгоритма Крускала с разреженными графами. В действительности, поиск-объединение может выполняться даже быстрее (см. *разд. 15.5*).

## 8.1.4. Разновидности остовных деревьев

Алгоритмы поиска минимальных остовных деревьев также можно использовать для решения родственных задач, в частности поиска следующих разновидностей остовных деревьев:

- *Максимальные остовные деревья.*

  Допустим, телефонная компания наняла подрядчика для прокладки телефонного кабеля между домами. Компания платит подрядчику с учетом протяженности проложенного кабеля. К сожалению, подрядчик оказался недобросовестным и пытается

получить от этого заказа дополнительный доход. В терминах теории графов для этого ему нужно создать наиболее «дорогое» остовное дерево. *Максимальное остовное дерево* любого графа можно найти, просто поменяв у всех ребер знак веса с плюса на минус и применив алгоритм Прима или Крускала. Максимальное по модулю остовное дерево в графе с отрицательным весом ребер окажется максимальным остовным деревом в первоначальном графе.

Большинство алгоритмов для графов не поддается адаптации под отрицательные числа с такой легкостью. Более того, алгоритмы поиска кратчайшего пути некорректно работают с отрицательными весами и *не возвращают* самый длинный путь при использовании подхода с отрицательными весами.

- *Остовные деревья с минимальным произведением весов ребер.*

  Допустим, нам нужно найти остовное дерево с минимальным произведением весов ребер, полагая, что вес всех ребер положительный. Так как $\lg(a \cdot b) = \lg(a) + \lg(b)$, то минимальное остовное дерево графа, в котором веса ребер были заменены их логарифмами, даст нам остовное дерево с минимальным произведением весов ребер первоначального графа.

- *Минимально критичное остовное дерево.*

  Иногда требуется найти остовное дерево, в котором наибольший вес ребра минимален среди всех возможных деревьев. Любое минимальное остовное дерево обладает этим свойством. Доказательство этого следует непосредственно из факта правильности алгоритма Крускала.

  Минимальные остовные деревья имеют интересные области применения, когда вес ребра представляет стоимость, пропускную способность и т. д. Менее эффективным, но концептуально более простым способом решения таких задач может быть удаление всех «тяжелых» ребер графа с проверкой полученного графа на связность. Эту проверку можно выполнить простым обходом в ширину или глубину.

Если все $m$ ребер графа имеют разный вес, то у такого графа существует единственное минимальное остовное дерево. В противном случае возвращаемое алгоритмом Прима или Крускала дерево определяется способом, который применяется в алгоритме для выбора одного из нескольких ребер с одинаковым весом.

Существуют две важные разновидности минимальных остовных деревьев, которые *не* поддаются решению способами, представленными в этом разделе:

- *Дерево Штейнера.*

  Допустим, нам нужно проложить проводку (ребра графа) между домами (вершины графа), но при выполнении этой задачи мы можем создавать дополнительные промежуточные пункты (вершины) в качестве общих соединений. Дерево, полученное в результате решения этой задачи, называется *минимальным деревом Штейнера* и рассматривается в *разд. 19.10*.

- *Остовное дерево с минимальной степенью вершин.*

  Предположим, нам нужно найти минимальное остовное дерево, у которого наибольшая степень вершины минимальна. Таким деревом будет простой путь с $n-2$ вершинами степени 2 и двумя конечными вершинами степени 1. Такой путь, который

проходит через каждую вершину только один раз, называется *гамильтоновым путем* и рассматривается в *разд. 19.5*.

## 8.2. История из жизни. И всё на свете — только сети

Однажды мне сообщили, что небольшая компания по тестированию печатных плат нуждается в консультации по алгоритмам. Вскоре я оказался в непримечательном здании за одним столом с президентом компании Integri-Test и ведущим техническим специалистом.

— Мы являемся ведущей компанией, поставляющей роботизированные устройства для тестирования печатных плат. Наши клиенты предъявляют очень высокие требования к надежности выпускаемых ими печатных плат. В частности, *перед тем* как устанавливать компоненты на печатную плату, они проверяют, что на ней нет разрывов в дорожках. Это означает, что им нужно убедиться, что каждая пара точек на плате, которые должны быть соединенными, в действительности соединены.

— И как вы выполняете это тестирование? — спросил я.

— Мы используем два роботизированных манипулятора, оснащенные электрическими щупами. Для проверки соединения между двумя точками манипуляторы подключают щупы к этим обеим точкам. Если дорожка между точками исправна, то щупы замыкают цепь. А для сетей мы фиксируем один из манипуляторов в одной точке, а вторым последовательно проверяем все остальные точки сети.

— Погодите! — вскричал я. — Что такое сеть на печатной плате?

— Печатные платы содержат несколько наборов точек, соединенных между собой посредством металлизированных дорожек (рис. 8.6, *а*). Вот эти наборы точек и соединяющие их дорожки мы и называем сетями. Часто сеть это просто две соединенные напрямую точки, а иногда может содержать 100–200 точек, как в случае с подключениями питания или заземления.

— Понятно. Значит, у вас есть список всех соединений между парами точек на печатной плате, и вы хотите проверить целостность этих соединяющих дорожек.

Он отрицательно покачал головой. — Не совсем так. Вход для нашей программы тестирования состоит только из контактных точек сети (рис. 8.6, *б*). Мы не знаем расположение отрезков соединяющей дорожки, но это и не требуется. Все, что нам нужно, — это проверить, что точки сети соединены вместе. Для этого один щуп приставляется к самой левой точке сети, а второй перемещается по всем остальным точкам, проверяя, что все они соединены с первой точкой. В таком случае они должны быть соединены друг с другом.

— Хорошо. Значит, правый манипулятор должен посетить все остальные точки сети. А каким образом вы определяете последовательность этих посещений?

На этот вопрос ответил технический специалист: «Мы сортируем точки слева направо и обходим их в этом порядке. Это хороший способ?»

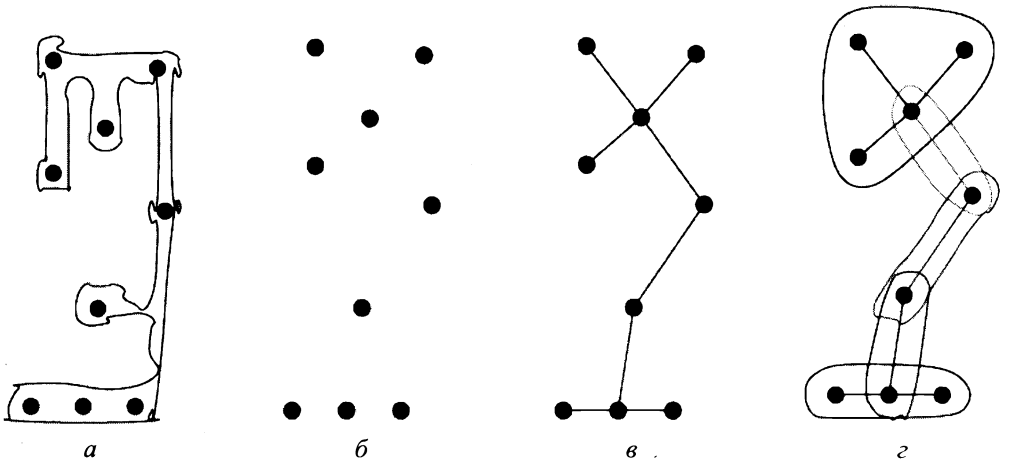



**Рис. 8.6.** Пример сети печатной платы: металлизированная соединяющая дорожка (*а*), контактные точки (*б*), минимальное остовное дерево контактных точек (*в*), контактные точки, разбитые на кластеры (*г*)

— Вы что-нибудь знаете о задаче коммивояжера? — спросил я.

Он был инженер-электрик, а не программист: «Нет. А что это?»

— Это название задачи, которую вы пытаетесь решить. Нужно посетить набор точек в порядке, который минимизирует время обхода. Алгоритмы для решения этой задачи всесторонне изучены. Для небольших сетей можно найти оптимальный маршрут методом полного перебора. А для больших сетей очень хорошее приближение оптимального маршрута можно найти посредством эвристических алгоритмов (для более подробной информации о решении задачи коммивояжера я бы показал им *разд. 19.4* этой книги, если бы она была у меня под рукой).

Президент компании что-то записал в своем блокноте, а потом нахмурился: «Хорошо. Допустим, вы сможете лучше упорядочить точки. Но проблема заключается не в этом. Когда мы наблюдаем за работой нашего робота, то иногда видим, что правый манипулятор доходит до самого правого края платы, в то время как левый манипулятор стоит неподвижно. Мне кажется, что было бы полезно разбивать сети на меньшие части, чтобы сбалансировать нагрузку на манипуляторы.

Я уселся поудобнее и стал обдумывать задачу. Оба манипулятора (левый и правый) решают взаимосвязанные задачи коммивояжера. Левый манипулятор перемещается от одной самой левой точки сети к другой, в то время как правый посещает все другие точки сети. Разбив каждую сеть на несколько меньших сетей, мы можем избежать ситуации, когда правый манипулятор проходит путь через всю плату. Кроме того, разбивка всей сети на части увеличит количество точек в задаче коммивояжера для левого манипулятора, вследствие чего каждое его перемещение также, скорее всего, будет небольшим.

— Вы правы. Разбивка больших сетей на несколько меньших должна повысить эффективность работы манипуляторов. Сети должны быть небольшого размера, как по количеству точек, так и по физической площади. Но нам необходимо гарантировать, что

проверив связность каждой малой сети, мы также проверим связность большой сети, членами которой они являются. Одной общей точки для двух малых сетей будет достаточно, чтобы доказать, что создаваемая объединением этих сетей большая сеть также связная, поскольку ток может протекать между любой парой точек.

Таким образом, нам нужно было решить задачу разбиения каждой сети на несколько меньших сетей, что является задачей кластеризации, или группирования. Для группирования часто применяются минимальные остовные деревья (см. *разд. 18.3*). Это и было решением нашей конкретной задачи кластеризации. В частности, мы смогли найти минимальное остовное дерево точек сети (рис. 8.6, *в*) и разбить его на несколько кластеров меньшего размера, разрывая любое слишком длинное ребро остовного дерева. Как можно видеть на рис. 8.6, *г*, каждый кластер точек будет иметь ровно одну общую точку с другим кластером, что обеспечивает связность, поскольку мы охватываем все ребра остовного дерева. Форма кластеров следует из расположения точек в сети. Если точки расположены на плате вдоль линии, то минимальным остовным деревом будет маршрут, а кластерами станут пары точек. А если точки расположены плотной группой, то этот кластер окажется хорошей площадкой для игры в классики правым манипулятором.

Я объяснил моим собеседникам идею создания минимального остовного дерева графа. Президент выслушал, что-то опять записал в своем блокноте и снова нахмурился.

— Мне нравится ваша идея кластеризации. Но минимальные остовные деревья определяются на графах, а у нас есть только точки. Где мы возьмем веса для ребер?

— О, мы можем рассматривать их как полный граф, в котором соединена каждая пара точек. А весом каждого ребра будет расстояние между двумя точками. Или не будет...?

Я опять начал размышлять. Стоимость ребра должна отражать время перемещения манипулятора между двумя точками. И хотя расстояние связано с временем перемещения, это не обязательно взаимозаменяемые понятия.

— У меня вопрос относительно вашего робота. Скорость перемещения манипулятора по горизонтали такая же, как и по вертикали?

Они подумали немного: «Конечно, такая же. Мы используем одинаковые двигатели для перемещения манипулятора по горизонтали и по вертикали. Так как оба двигателя работают независимо друг от друга, манипулятор можно перемещать одновременно и по горизонтали, и по вертикали.

— То есть перемещение на один фут влево и один фут вверх занимает точно такое же время, как перемещение на один фут только влево? Это означает, что вес каждого ребра должен соответствовать не евклидову расстоянию между двумя точками, а наибольшему расстоянию между ними либо по горизонтали, либо по вертикали. Это называется метрикой $L_\infty$, но мы можем зафиксировать ее, изменяя вес ребер в графе. Происходит ли еще что-нибудь необычное с вашими роботами? — спросил я.

— Роботу нужно некоторое время, чтобы развить рабочую скорость. Также, наверное, нужно принять во внимание время на ускорение и замедление движения манипулятора.

— Абсолютно верно. Чем точнее мы смоделируем перемещение манипулятора между двумя точками, тем лучше будет наше решение. И теперь у нас есть очень четкая формулировка решения. Давайте закодируем его и посмотрим, насколько оно хорошо.

Они немного засомневались, что этот подход принесет какую-нибудь пользу, но согласились подумать. Несколько недель спустя они позвонили мне и сообщили, что новый алгоритм за счет небольшого увеличения вычислений сокращает дистанцию маршрута примерно на 30% по сравнению с их первоначальным подходом. Но т. к. их тестовое оборудование стоило 200 тыс. долларов, а персональный компьютер — всего лишь 2 тыс. долларов, то это был замечательный компромисс. Кроме того, им показалось весьма выгодным, что в случае тестирования партии одинаковых плат алгоритм придется выполнять только один раз.

Ключевой идеей в этом случае было моделирование задачи посредством классических алгоритмов для решения задач на графах. Я понял, что передо мной задача коммивояжера, как только разговор зашел о минимизации протяженности перемещений манипулятора. Когда я выяснил, что для проверки связности точек они неявно использовали звездообразное остовное дерево, было естественным задать вопрос, не улучшило бы производительность минимальное остовное дерево. Эта идея проложила путь к идее кластеризации, т. е. разбиению каждой сети на несколько меньших сетей. Наконец, внимательный подход к разработке метрик расстояний, чтобы точно смоделировать особенности манипулятора, позволил нам внедрить в решение сложные свойства (такие как ускорение и замедление манипулятора), не изменяя фундаментальной модели графа или структуры алгоритма.

#### *Подведение итогов*

Большинство приложений с использованием графов можно свести к стандартным задачам на графах, к которым можно применить хорошо известные алгоритмы. Это такие задачи, как построение минимальных остовных деревьев, кратчайших путей и прочие задачи, представленные в каталоге задач этой книги.

## 8.3. Поиск кратчайшего пути

*Путь* (path) — это последовательность ребер, соединяющих две вершины. Два узла любой дороги или социальной сети обычно можно соединить громадным количеством возможных путей. Наиболее интересным из этих путей станет, скорее всего, путь, который минимизирует сумму весов ребер, т. е. кратчайший путь, т. к. он определит наиболее быстрый маршрут или наиболее близкое родство между двумя узлами.

В невзвешенном графе кратчайший путь от вершины $s$ к вершине $t$ можно определить посредством обхода в ширину с исходной точкой в вершине $s$. Дерево обхода в ширину предоставляет путь, состоящий из минимального количества ребер. Это кратчайший путь, если все ребра имеют одинаковый вес.

Но во взвешенных графах обход в ширину *не* предоставляет кратчайшего пути. В этом типе графа кратчайший взвешенный путь может потребовать большего количества ребер, точно так же, как и наиболее быстрый маршрут от дома к офису может проходить по нескольким отрезкам второстепенных дорог вместо одного или двух прямых отрезков главной дороги, как показано на рис. 8.7.

В этом разделе мы рассмотрим два разных алгоритма поиска кратчайшего пути во взвешенных графах.

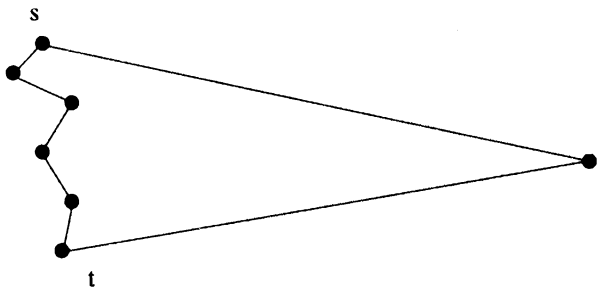



**Рис. 8.7.** Кратчайший путь от точки *s* к точке *t* может проходить через несколько промежуточных точек вместо того, чтобы использовать наименьшее количество ребер

## 8.3.1. Алгоритм Дейкстры

Алгоритм Дейкстры является предпочтительным методом поиска кратчайших путей в графах со взвешенными ребрами и/или вершинами. Алгоритм находит кратчайший путь от заданной начальной вершины *s* ко всем другим вершинам графа, включая требуемую конечную вершину *t*.

Допустим, что кратчайший путь от вершины *s* к вершине *t* графа *G* проходит через определенную промежуточную вершину *x*. Очевидно, что наилучший путь между этими вершинами должен прежде всего содержать кратчайший путь от вершины *s* к вершине *x*, ибо в противном случае можно было бы улучшить путь от *s* к *t*, предложив более короткий начальный путь от *s* к *x*. Таким образом, прежде чем искать кратчайший путь от начальной вершины *s* к конечной вершине *t*, нужно найти кратчайший путь от начальной вершины *s* к промежуточной вершине *x*.

Алгоритм Дейкстры работает поэтапно, находя на каждом этапе кратчайший путь от вершины *s* к некой *новой* вершине. Говоря конкретно, вершина *x* такова, что сумма $dist(s, v_i) + w(v_i, x)$ минимальна для всех необработанных *вершин* $v_i$, где $w(a, b)$ — вес ребра между вершинами *a* и *b*, а $dist(a, b)$ — длина кратчайшего пути между ними.

Здесь напрашивается стратегия, аналогичная динамическому программированию. Кратчайший путь от вершины *s* к самой себе является тривиальным, поэтому $dist(s, s) = 0$[1]. Если $(s, y)$ является самым легким ребром, входящим в вершину *s*, тогда $dist(s, y) = w(s, y)$. Определив кратчайший путь к вершине *x*, мы проверяем все исходящие из нее ребра, чтобы узнать, не существует ли более короткого пути из начальной вершины *s* к какой-либо неизвестной вершине через вершину *x*. Псевдокод этого алгоритма представлен в листинге 8.9.

**Листинг 8.9. Псевдокод алгоритма Дейкстры**

```
ShortestPath-Dijkstra(G, s, t)
    known = {s}
```

[1] В действительности это так только в случае графов, не содержащих ребер с отрицательным весом. По этой причине в последующем рассмотрении мы предполагаем, что все ребра имеют положительным вес.

```
for each vertex v in G, dist[v] = ∞
dist[s] = 0
for each edge (s, v), dist[v] = w(s, v)
last = s

while (last ≠ t)
    select v_next, неизвестная вершина, минимизирующая dist[v]
    for each edge (v_next, x), dist[x] = min[dist[x],
                          dist[v_next] + w(v_next, x) ]
    last = v_next
    known = known ∪ {v_next}
```

Основная идея алгоритма Дейкстры очень похожа на основную идею алгоритма Прима. В каждом цикле мы добавляем одну вершину к дереву вершин, для которых мы *знаем* кратчайший путь из вершины *s*. Так же как и в алгоритме Прима, мы сохраняем информацию о наилучшем пути на текущий момент для всех вершин вне дерева и вставляем их в дерево в порядке возрастания веса.

По сути, *единственная* разница между алгоритмом Дейкстры и алгоритмом Прима состоит в том, как они оценивают привлекательность каждой вершины вне дерева. В задаче поиска минимального остовного дерева мы стремились минимизировать вес следующего возможного ребра дерева. А при поиске кратчайшего пути нам нужна также информация о ближайшей к вершине *s* вершине вне дерева — т. е. кроме веса нового ребра мы хотим знать еще *и расстояние* от вершины *s* к смежной с ней вершине дерева.

### Реализация

Псевдокод алгоритма Дейкстры скрывает его близкое сходство с алгоритмом Прима. В листинге 8.10 приводится реализация алгоритма Дейкстры, которая, по сути, является реализацией алгоритма Прима, в которой изменены всего лишь четыре строчки кода (отмечены в листинге комментариями), одна из которых — просто имя функции.

**Листинг 8.10. Реализация алгоритма Дейкстры**

```
int dijkstra(graph *g, int start) {
    int i;                  /* Счетчик */
    edgenode *p;            /* Временный указатель */
    bool intree[MAXV+1];    /* Вершина уже в дереве? */
    int distance[MAXV+1];   /* Стоимость добавления в дерево */
    int v;                  /* Текущая вершина для обработки */
    int w;                  /* Кандидат на следующую вершину */
    int dist;               /* Наименьшая стоимость расширения дерева */
    int weight = 0;         /* Вес дерева */

    for (i = 1; i <= g->nvertices; i++) {
        intree[i] = false;
        distance[i] = MAXINT;
        parent[i] = -1;
    }
```

```
    distance[start] = 0;
    v = start;
    while (!intree[v]) {
        intree[v] = true;
        if (v != start) {
            printf("edge (%d,%d) in tree \n",parent[v],v);
            weight = weight + dist;
        }
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            if (distance[w] > (distance[v]+p->weight)) {  /* ИЗМЕНЕНИЕ */
                distance[w] = distance[v]+p->weight;      /* ИЗМЕНЕНИЕ */
                parent[w] = v;                            /* ИЗМЕНЕНИЕ */
            }
            p = p->next;
        }

        dist = MAXINT;
        for (i = 1; i <= g->nvertices; i++) {
            if ((!intree[i]) && (dist > distance[i])) {
                dist = distance[i];
                v = i;
            }
        }
    }

    return(weight);
}
```

Этот алгоритм определяет минимальное остовное дерево с корнем в вершине *s*. Для невзвешенных графов это будет дерево обхода в ширину, но, в общем, алгоритм предоставляет кратчайший путь от вершины *s* ко всем остальным вершинам, а не только к вершине *t*.

### Анализ

Какова временная сложность алгоритма Дейкстры? В том виде, в котором он реализован здесь, его временная сложность равна $O(n^2)$. Это точно такое же время исполнения, как и для алгоритма Прима, что не удивительно, — ведь за исключением дополнительного условия, это *точно такой же* алгоритм, как и алгоритм Прима.

Длина кратчайшего пути от начальной вершины `start` к заданной вершине *t* равна точно значению `distance[t]`. Каким образом мы можем найти сам путь с помощью алгоритма Дейкстры? Точно так же, как в процедуре `find_path()` поисков в ширину и длину (см. *разд. 7.6.2*), мы следуем обратным указателям на родительские узлы от вершины *t*, пока не дойдем до начальной вершины (или пока не получим –1, если такого пути не существует).

Алгоритм Дейкстры работает правильно только на графах, в которых нет ребер с отрицательным весом. Дело в том, что при построении пути может встретиться ребро с отрицательным весом, настолько большим по модулю, что оно полностью изменит опти-

мальный путь от вершины $s$ к какой-либо другой вершине, которая уже включена в дерево. Образно говоря, самым выгодным путем к соседу по лестничной клетке может оказаться путь через банк на другом конце города, если этот банк выдает за каждое посещение достаточно большое вознаграждение, делающее такой маршрут выгодным. Если банк в нашем примере не ограничивает вознаграждение одноразовой выплатой, то выгода от его бесконечного посещения заставит вас *навсегда* отказаться от идеи навестить соседа.

К счастью, в большинстве приложений ребра с отрицательным весом не используются, что делает это ограничение на пригодность алгоритма Дейкстры чисто теоретическим. Рассматриваемый далее алгоритм Флойда — Уоршелла работает правильно при наличии ребер с отрицательным весом, но при условии отсутствия *циклов* с отрицательной стоимостью, которые значительно искажают структуру кратчайшего пути.

**ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ:**

### Кратчайший путь с учетом веса вершин

**ЗАДАЧА**. Допустим, что нам нужно разработать эффективный алгоритм для поиска самого дешевого пути в ориентированном графе со взвешенными вершинами, а не со взвешенными ребрами. То есть стоимостью пути от вершины $x$ к вершине $y$ является сумма весов всех вершин в пути.

**РЕШЕНИЕ.** Естественным решением будет адаптация алгоритма для графов со взвешенными ребрами (например, алгоритма Дейкстры) для этой ситуации. Очевидно, что такой подход работоспособен. Мы просто заменим все упоминания веса ориентированного ребра $(x, y)$ на вес его конечной вершины $y$. Эту информацию можно хранить в массиве и обращаться к ней по мере надобности.

Но лично я предпочитаю вместо этого модифицировать граф таким образом, чтобы алгоритм Дейкстры дал желаемый результат. Для этого мы установим в качестве веса ориентированного ребра $(i, j)$ вес вершины $j$. Применение алгоритма Дейкстры на таком графе дает требуемый результат. Стремитесь *разрабатывать графы, а не алгоритмы*, как рекомендуется в *разд. 8.7*.

Эту технику можно распространить на другие задачи, включая такие, в которых вес имеется как у ребер, так и у вершин. ■

## 8.3.2. Кратчайшие пути между всеми парами вершин

Допустим, что нам нужно найти «центральную» вершину графа, т. е. вершину с кратчайшими путями ко всем остальным вершинам. Практическим примером такой задачи может быть выбор места для открытия офиса. Или пусть вам надо узнать *диаметр* графа, т. е. максимальное кратчайшее расстояние между всеми парами вершин. Практическим примером такой задачи может быть определение максимального временного интервала для доставки письма или сетевого пакета. В этих случаях нужно найти кратчайшие пути между всеми парами вершин графа.

Одно из возможных решений этой задачи — последовательное применение алгоритма Дейкстры для каждой из возможных $n$ начальных вершин. Но более правильно задей-

ствовать алгоритм Флойда — Уоршелла (алгоритм кратчайшего пути для всех пар), чтобы создать матрицу расстояний размером $n \times n$ из исходной матрицы весов графа.

Алгоритм Флойда — Уоршелла лучше всего использовать на матрицах смежности (adjacency matrix). В этом нет ничего необычного, поскольку нам требуется сохранить все $n$ попарных расстояний в любом случае. В листинге 8.11 приводится определение структуры для хранения требуемой информации.

**Листинг 8.11. Определение типа матрицы смежности**

```
typedef struct {
   int weight[MAXV+1][MAXV+1];      /* Информация о смежности/весе */
   int nvertices;                   /* Количество вершин в графе */
} adjacency_matrix;
```

Под тип `adjacency_matrix` выделяется память для матрицы максимально большого размера. Кроме того, в нем предусмотрено поле для хранения информации о количестве вершин в графе.

Критическим вопросом в реализации матрицы смежности является способ обозначения отсутствующих в графе ребер. Обычно для обозначения присутствующих ребер невзвешенных графов используется 1, а отсутствующих — 0. Но если эти числа обозначают вес ребер, то такое обозначение дает неверный результат, поскольку отсутствующие ребра рассматриваются как «бесплатный» путь между вершинами. Вместо этого отсутствующие ребра нужно инициализировать значением `MAXINT`. Таким образом мы можем и проверять наличие ребра, и автоматически игнорировать его при поиске кратчайшего пути.

Кратчайший путь между двумя вершинами графа можно определить несколькими способами. Алгоритм Флойда — Уоршелла начинает работу с присвоения вершинам графа номеров от 1 до $n$. Эти номера используются не для маркировки вершин, а для их упорядочивания. Определим $W[i,j]^k$ как длину кратчайшего пути от вершины $i$ к вершине $j$, в котором используются в качестве промежуточных вершин только вершины, пронумерованные 1, 2, ..., $k$.

Что это означает? Когда $k = 0$, то промежуточные вершины недопустимы, поэтому единственными разрешенными путями являются первоначальные ребра графа. Таким образом, матрица кратчайших путей для всех пар вершин состоит из первоначальной матрицы смежности. Будет выполнено $n$ циклов, где в $k$-м цикле разрешается использовать только первые $k$ вершин в качестве возможных промежуточных вершин на пути между каждой парой вершин $x$ и $y$.

В каждом последующем цикле добавляется одна новая возможная промежуточная вершина, что расширяет набор возможных кратчайших путей. Использование $k$-й вершины в качестве конечной помогает только в том случае, если существует кратчайший путь, который проходит через эту вершину, поэтому

$$W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1}).$$

Правильность этого выражения не совсем очевидна с первого взгляда, поэтому я рекомендую хорошо разобраться в нем, чтобы убедиться в этом. Это замечательный пример

парадигмы разработки алгоритмов для динамического программирования, которой посвящена *глава 10*. Но, как видно из листинга 8.12, реализация алгоритма Флойда — Уоршелла достаточно проста.

**Листинг 8.12. Реализация алгоритма Флойда — Уоршелла**

```
void floyd(adjacency_matrix *g) {
    int i, j;            /* Счетчики измерений */
    int k;               /* Счетчик промежуточных вершин */
    int through_k;       /* Длина пути через вершину k */

    for (k = 1; k <= g->nvertices; k++) {
        for (i = 1; i <= g->nvertices; i++) {
            for (j = 1; j <= g->nvertices; j++) {
                through_k = g->weight[i][k]+g->weight[k][j];
                if (through_k < g->weight[i][j]) {
                    g->weight[i][j] = through_k;
                }
            }
        }
    }
}
```

Время исполнения алгоритма Флойда — Уоршелла равно $O(n^3)$, что ничем не лучше, чем время исполнения $n$ вызовов алгоритма Дейкстры. Но циклы этого алгоритма такие сжатые, а сама программа такая короткая, что на практике он оказывается более эффективным. Алгоритм Флойда — Уоршелла примечателен тем, что это один из немногих алгоритмов работы с графами, которые работают лучше на матрицах смежности, чем на списках смежности.

По результату работы алгоритма Флойда — Уоршелла в его текущей форме нельзя воссоздать фактический кратчайший путь между любой парой вершин. Но эти пути можно получить, если мы сохраним матрицу родителей $P$, содержащую наш выбор последней промежуточной вершины и используемую для каждой пары вершин $(x, y)$. Пусть это будет вершина $k$. Кратчайшим путем от вершины $x$ к вершине $y$ будет конкатенация кратчайшего пути от вершины $x$ к вершине $k$ и кратчайшего пути от вершины $k$ к вершине $y$, которую можно воссоздать рекурсивным способом по матрице $P$. Но обратите внимание, что в большинстве задач поиска кратчайшего пути между всеми парами точек требуется только полученная матрица расстояний. Для таких приложений и предназначен алгоритм Флойда — Уоршелла.

## 8.3.3. Транзитивное замыкание

Алгоритм Флойда — Уоршелла имеет еще одну важную область применения — вычисление *транзитивного замыкания* (transitive closure). При анализе ориентированного графа часто требуется знать, какие вершины достижимы из того или иного узла.

В качестве примера рассмотрим граф шантажиста, в котором наличие ориентированного ребра $(i, j)$ означает, что лицо $i$ обладает достаточным компроматом на лицо $j$ и мо-

жет заставить его сделать все, что угодно. Для участия в специальном проекте вы хотите нанять одного из этих *n* людей, обладающего наибольшим потенциалом шантажа.

Простым решением было бы нанять шантажиста, представленного вершиной с наивысшей полустепенью исхода, но лучше всего выбрать шантажиста с компроматом на наибольшее число людей — т. е. вершину со связями с наибольшим количеством других вершин. Например, Стив может шантажировать напрямую только Майкла, но если Майкл имеет компромат на всех прочих, тогда именно Стив окажется лучшим выбором для нашего проекта.

Вершины, достижимые из любой другой вершины, можно найти посредством обхода в ширину или глубину. Но полное множество взаимоотношений можно вычислить с помощью алгоритма кратчайшего пути между всеми парами вершин. Если после исполнения алгоритма Флойда — Уоршелла длина кратчайшего пути между вершиной *i* и вершиной *j* равняется `MAXINT`, то можно быть уверенным, что между этими вершинами прямого пути не существует. Любые две вершины, у которых вес ребра меньше `MAXINT`, достижимы, как и в смысле теории графов, так и в смысле описанного проекта.

Транзитивное замыкание рассматривается более подробно в *разд. 18.5*.

## 8.4. История из жизни. Печатаем с помощью номеронабирателя

В составе группы специалистов я попал на экскурсию в компанию Periphonics, которая в то время была лидером в области создания телефонных систем речевого взаимодействия. Это более интеллектуальные системы, чем системы типа «Нажмите кнопку 1 для перехода в меню», «Нажмите кнопку 2, если вы не нажали 1», которые так раздражают нас в повседневной жизни. Гид настолько далеко зашел в превознесении достоинств их продукта, что вывел из себя одного из менее сдержанных членов нашей делегации.

— Вводить текст с помощью номеронабирателя? Не смешите меня! — подал он голос из задних рядов группы. — Я *ненавижу* такой способ ввода текста. Всегда, когда я звоню в свою маклерскую контору, чтобы узнать биржевой курс, какая-то машина велит мне ввести трехбуквенный код. Что еще хуже, чтобы указать, какую именно из трех букв, нанесенных на кнопку, ввести, нужно нажать на нее два раза. Я нажимаю кнопку 2, а машина мне говорит: «Нажмите 1 для A, Нажмите 2 для B, Нажмите 3 для C». Это ужасно, если вас интересует мое мнение.

— Возможно, не нужно нажимать две кнопки, чтобы ввести одну букву, — подключился я к разговору. — Может быть, система сумеет вычислить требуемую букву по контексту.

— Когда вводишь трехбуквенный биржевой код акции, контекста маловато.

— Конечно, но если вводить предложения, то контекста будет предостаточно. Спорю, что мы смогли бы правильно восстановить текст, вводимый с панели телефона по одной кнопке на букву.

Сотрудник Periphonics бросил на меня равнодушный взгляд и продолжил экскурсию. Но мне эта идея запала в голову, и когда я вернулся в свой офис, то решил попытаться воплотить ее в жизнь.

При вводе с номеронабирателя не все буквы можно ввести одинаковым способом. Более того, даже не все буквы вообще можно ввести, т. к. на стандартном американском телефоне буквы Q и Z не обозначены. Поэтому было условлено, что буквы Q, Z и пробел будут набираться нажатием кнопки звездочки <*>. Для распознавания вводимого с номеронабирателя текста можно было бы воспользоваться частотой использования букв алфавита. Например, если нажата кнопка <3>, то существует большая вероятность, что имеется в виду буква E, а не находящиеся на этой же кнопке буквы D или F. Наша первая попытка предсказать вводимый текст была основана на частоте вхождения каждого из трех символов (триграмм) в строку текста. Но результаты были обескураживающими. В частности, метод триграмм оказался вполне пригодным для преобразования вводимого текста в бессмыслицу, но совсем неприемлемым для его понятного транскрибирования на английском языке.

Одна причина этому была очевидной — алгоритм не знает абсолютно ничего об английских словах. Если применить его совместно со словарем, то мы, возможно, получим какой-то результат. Но в словаре может быть несколько разных слов для одной и той же последовательности кнопок номеронабирателя. В качестве крайнего примера: последовательность 22737 может означать одиннадцать разных английских слов, включая *cases*, *cares*, *cards*, *capes*, *caper* и *bases*. В нашей следующей попытке для последовательностей кнопок с несколькими возможными словами мы выводили буквы, в правильности которых не было сомнений, а остальные угадывали методом триграмм.

Результаты этого подхода также были ужасными. Большинство слов в тексте создавались вследствие неоднозначного сопоставления кодов нескольким словам из словаря. Нам нужно было найти какой-то способ различать слова, набираемые одинаковой последовательностью кнопок. Можно было бы попробовать учитывать относительную частоту использования каждого слова, но здесь также было бы слишком много ошибок.

На этом этапе я подключил к проекту Гаральда Рау (Harald Rau), который оказался прекрасным помощником. Прежде всего, это был смышленый и упорный аспирант. Кроме того, поскольку его родным языком был немецкий, он верил всему, что я говорил ему об английской грамматике. В результате Гаральд создал алгоритм реконструкции слов по нажатым клавишам (рис. 8.8).

Программа обрабатывала по одному предложению за раз, идентифицируя все слова, совпадающие с каждой последовательностью введенного кода. Принципиальную трудность представляло встраивание в нее грамматических правил.

— Хорошая информация о грамматике и о частоте использования слов имеется в большой базе данных Brown Corpus. Она содержит тысячи типичных английских предложений, проанализированных по частям речи. Но как мы можем учесть всю эту информацию в нашей программе? — спросил Гаральд.

— Давай будем рассматривать это как задачу на графах, — предложил я.

— *Задачу на графах?* Где же здесь графы?

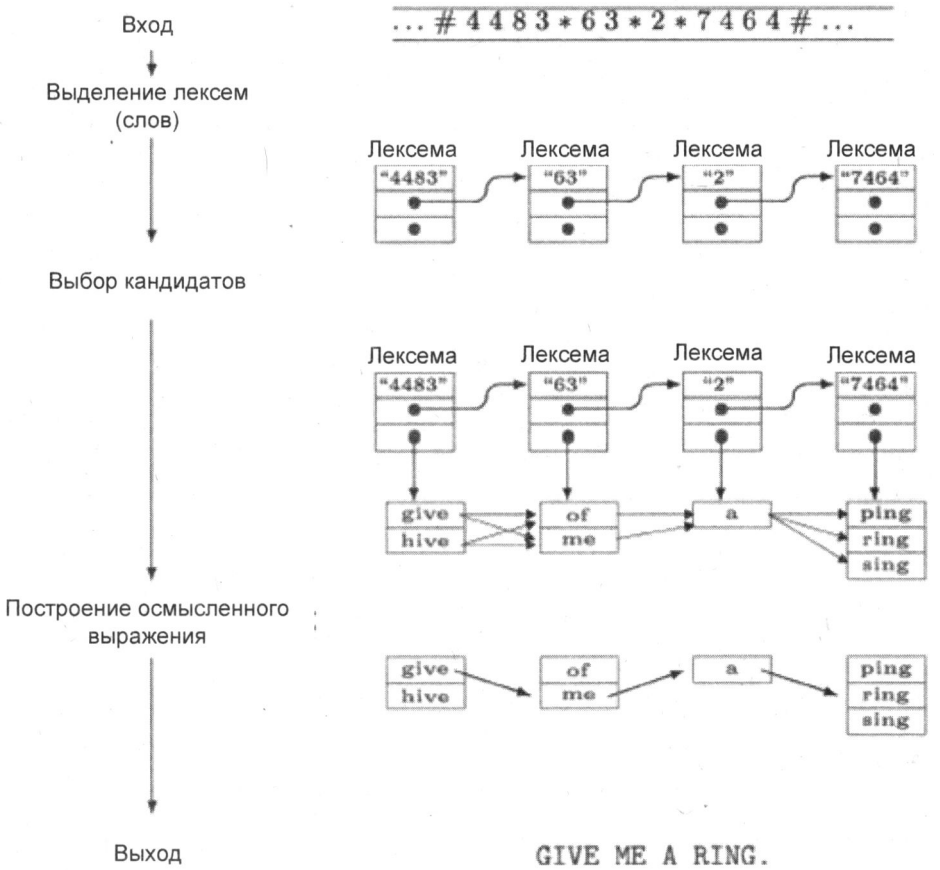



**Рис. 8.8.** Этапы восстановления слов из последовательностей нажатых кнопок номеронабирателя

— Предложение можно рассматривать как последовательность лексем, каждая из которых представляет слово в предложении. Для каждой лексемы имеется соответствующий список слов, из которого нужно выбрать правильное. Как это сделать? Каждое возможное предложение, т. е. комбинацию слов, можно рассматривать как путь в графе, каждая вершина которого представляет одно слово из полного набора возможных слов. Каждое возможное $i$-е слово будет соединено ребром с каждым возможным $(i + 1)$ словом. Самым дешевым путем по этому графу будет наилучшая интерпретация предложения.

— Но все пути выглядят одинаково: они имеют одинаковое количество ребер. Ага, теперь я вижу! Чтобы сделать пути разными, нам нужно присвоить ребрам вес.

— Совершенно верно! Стоимость ребра будет отражать вероятность нашего выбора соединяемой им пары слов. Эту стоимость можно определять по тому, как часто та или иная пара слов встречается в базе данных. Или вес можно присваивать с учетом частей речи. Возможно, существительные реже соседствуют с другими существительными, чем с глаголами.

— Отслеживать статистику пар слов будет трудно, поскольку возможных пар так много! Но мы знаем частоту использования каждого слова. Как мы можем учесть этот фактор в нашей программе?

— Можно сделать так, что цена за проход через определенную вершину станет зависеть от частоты употребления этого слова. В таком случае самый короткий путь через граф будет самым лучшим предложением.

— Но как нам определить относительную важность каждого из этих факторов?

— Попробуй реализовать то, что тебе кажется естественным, а потом можно экспериментировать.

Алгоритм поиска кратчайшего пути, разработанный Гаральдом по этим принципам, показан на рис. 8.9.

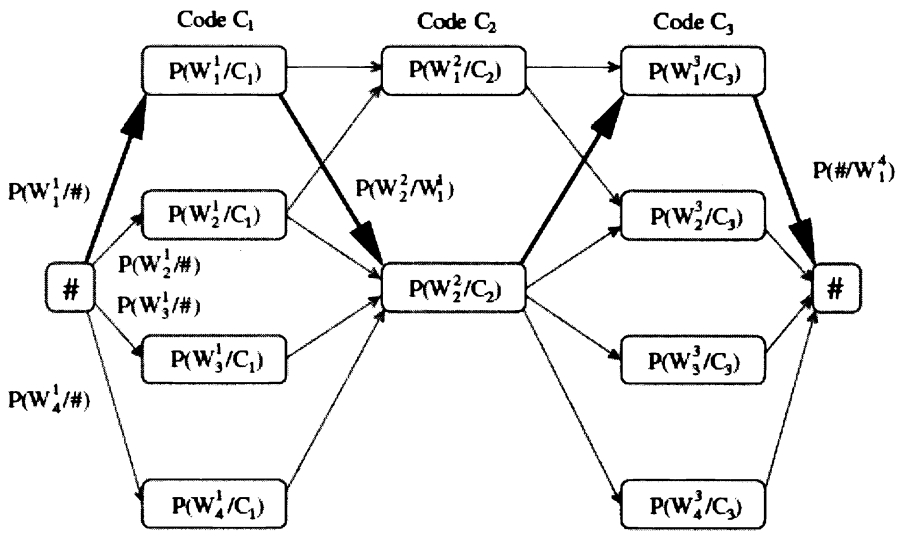



**Рис. 8.9.** Путь с наименьшей стоимостью представляет самую лучшую интерпретацию предложения

Программа со встроенными грамматическими правилами и статистическими условиями работала прекрасно. Оцените ее воспроизведение Геттисбергского послания[2], введенного с номеронабирателя телефона, где ошибки воспроизведения выделены полужирным шрифтом и подчеркиванием (рис. 8.10).

Хотя программа все еще допускала некоторые ошибки, обычно она угадывала правильно около 99% всех символов. Такой результат, несомненно, является достаточно хорошим для многих приложений. В компании Periphonics определенно были такого же мнения, поскольку они лицензировали нашу программу для использования

[2] Геттисбергская речь Авраама Линкольна — одна из известнейших речей в истории Соединенных Штатов Америки. Президент произнес ее 19 ноября 1863 года при открытии Национального солдатского кладбища в Геттисберге, штат Пенсильвания. — *Прим. ред.*

в своих продуктах. Скорость воссоздания текста оказалась быстрее, чем ввод его с клавиатуры номеронабирателя.

Условия для многих задач распознавания закономерностей можно сформулировать естественным образом в виде задач поиска кратчайшего пути в графе. Для решения этих задач хорошо подходит алгоритм Витерби, который широко применяется в системах распознавания речи и рукописи. Алгоритм Витерби, по сути, решает задачу поиска кратчайшего пути в бесконтурном ориентированном графе. Таким образом, часто будет правильным попытаться сформулировать поставленную задачу в терминах теории графов.

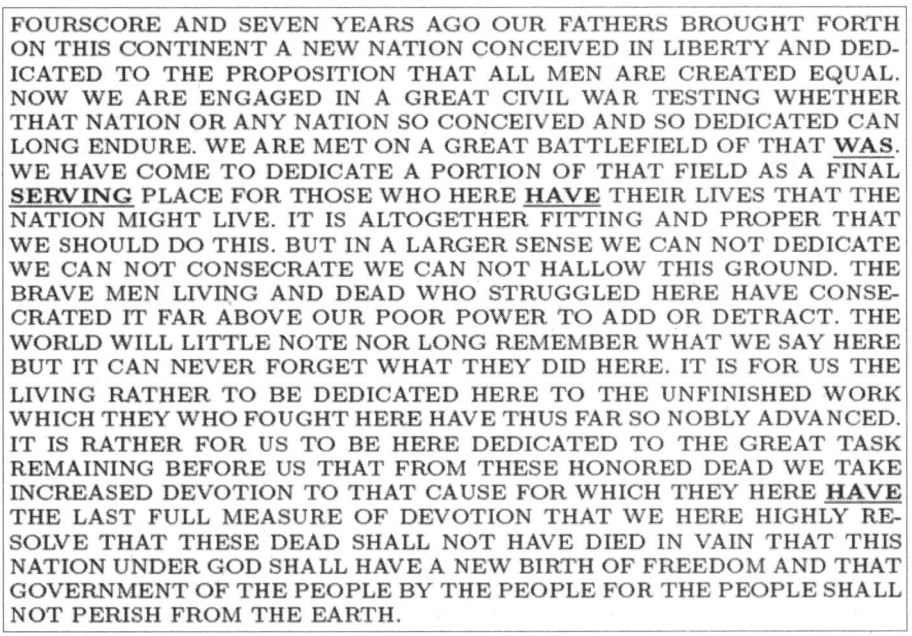

FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL. NOW WE ARE ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT WAS. WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL SERVING PLACE FOR THOSE WHO HERE HAVE THEIR LIVES THAT THE NATION MIGHT LIVE. IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO THIS. BUT IN A LARGER SENSE WE CAN NOT DEDICATE WE CAN NOT CONSECRATE WE CAN NOT HALLOW THIS GROUND. THE BRAVE MEN LIVING AND DEAD WHO STRUGGLED HERE HAVE CONSECRATED IT FAR ABOVE OUR POOR POWER TO ADD OR DETRACT. THE WORLD WILL LITTLE NOTE NOR LONG REMEMBER WHAT WE SAY HERE BUT IT CAN NEVER FORGET WHAT THEY DID HERE. IT IS FOR US THE LIVING RATHER TO BE DEDICATED HERE TO THE UNFINISHED WORK WHICH THEY WHO FOUGHT HERE HAVE THUS FAR SO NOBLY ADVANCED. IT IS RATHER FOR US TO BE HERE DEDICATED TO THE GREAT TASK REMAINING BEFORE US THAT FROM THESE HONORED DEAD WE TAKE INCREASED DEVOTION TO THAT CAUSE FOR WHICH THEY HERE HAVE THE LAST FULL MEASURE OF DEVOTION THAT WE HERE HIGHLY RESOLVE THAT THESE DEAD SHALL NOT HAVE DIED IN VAIN THAT THIS NATION UNDER GOD SHALL HAVE A NEW BIRTH OF FREEDOM AND THAT GOVERNMENT OF THE PEOPLE BY THE PEOPLE FOR THE PEOPLE SHALL NOT PERISH FROM THE EARTH.

**Рис. 8.10.** Результат ввода Геттисбергского послания с номеронабирателя телефона

## 8.5. Потоки в сетях и паросочетание в двудольных графах

Графы со взвешенными ребрами можно рассматривать как трубопроводную сеть, в которой вес ребра определяет *пропускную способность* соответствующего отрезка трубопровода. В свою очередь, пропускную способность можно рассматривать как функцию поперечного сечения трубы. Широкая труба может пропускать 10 единиц потока (т. е. объем материала) за определенное время, тогда как более узкая труба — только 5 единиц за то же самое время. В *задаче о потоках в сетях* требуется определить максимальный объем потока, который можно пропустить между вершинами $s$ и $t$ взвешенного графа $G$, соблюдая ограничения на максимальную пропускную способность каждого ребра-трубы.

## 8.5.1. Паросочетание в двудольном графе

В то время как задача о потоках в сетях представляет самостоятельный интерес, ее основная ценность состоит в решении других важных задач на графах. Классическим примером является паросочетание в двудольном графе. *Паросочетанием* графа $G = (V, E)$ называется такое подмножество ребер $E' \subset E$, в котором никакие два ребра не имеют общей вершины. При этом вершины группируются попарно таким образом, что каждая вершина принадлежит не более чем одной такой паре. Двудольный граф с максимальным паросочетанием показан на рис. 8.11, *а*, а соответствующий экземпляр потоков в сети с максимальным потоком от $s$ к $t$ — на рис. 8.11, *б*.

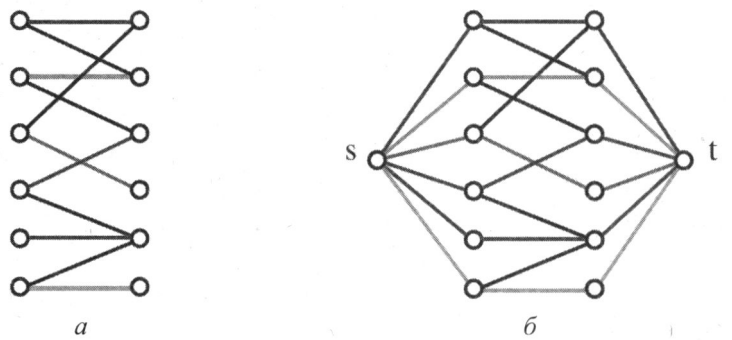



**Рис. 8.11.** Двудольный граф с максимальным паросочетанием (*а*) и соответствующий экземпляр потоков в сети (*б*)

Граф $G$ называется *двудольным*, если его вершины можно разделить на два множества: $L$ и $R$ — таким образом, чтобы все ребра графа имели одну вершину в множестве $L$, а другую — в множестве $R$. Многие естественные графы являются двудольными. Например, вершины одного класса могут представлять задания, требующие исполнения, а остальные вершины — людей, которые могут выполнить эти задания. Наличие ребра $(j, p)$ означает, что задание $j$ может быть выполнено человеком $p$. Или же часть вершин может представлять юношей, а другая — девушек, а ребра — совместимые гетеросексуальные пары. Паросочетания в этих графах поддаются естественному толкованию как рабочие задания или традиционные бракосочетания. Такие паросочетания подробно обсуждаются в *разд. 18.6*.

Парообразования с максимальной кардинальностью в двудольном графе можно с легкостью найти, используя потоки в сети. Для этого создаем вершину-*исток* $s$, которая соединена со всеми другими вершинами в подмножестве $L$ взвешенными ребрами, каждое весом 1. Потом создаем вершину-*сток* $t$, которая соединена со всеми другими вершинами в подмножестве $R$ взвешенными ребрами, каждое весом 1. Наконец, присваиваем каждому ребру центрального двудольного графа $G$ вес 1. Теперь максимальный возможный поток из вершины $s$ в вершину $t$ определяет максимальное паросочетание в графе $G$. Всегда можно найти поток такого же объема, как и паросочетание, используя только ребра паросочетания и соединенные ими истоки и стоки. Кроме этого, решение, дающее поток большего объема, невозможно. Ведь не можем же мы пропускать через любую вершину больше чем одну единицу потока.

## 8.5.2. Вычисление потоков в сети

Традиционные алгоритмы работы с потоками в сети основаны на идее *увеличивающих путей*, которая состоит в поиске пути с положительной пропускной способностью от вершины $s$ к вершине $t$ и добавления его к потоку. Можно доказать, что поток в сети является оптимальным тогда и только тогда, когда в ней не имеется увеличивающего пути. Так как каждое добавление пути увеличивает поток, повторяем процесс поиска до тех пор, пока больше не останется таких путей, в итоге должен быть найден глобальный максимум.

Ключевой структурой является *граф остаточного потока* (residual flow graph) $R(G, f)$, где $G$ — граф ввода, веса которого представляют пропускную способность, а $f$ — массив потоков через $G$. Ориентированный граф $R(G, f)$ с взвешенными ребрами содержит те же самые вершины, что и граф $G$. Для каждого ребра $(i, j)$ в графе $G$, которое обладает пропускной способностью $c(i, j)$ и имеет поток $f(i, j)$, граф $R(G, f)$ может содержать следующие два ребра:

- ребро $(i, j)$ с весом $c(i, j) - f(i, j)$, если $c(i, j) - f(i, j) > 0$;
- ребро $(j, i)$ с весом $f(i, j)$, если $f(i, j) > 0$.

Вес ребра $(i, j)$ в графе остаточного потока указывает точный объем дополнительного потока, который можно направить от вершины $i$ к вершине $j$. Путь в графе остаточного потока от вершины $s$ к вершине $t$ подразумевает, что от первой вершины ко второй можно пропустить дополнительный поток. Наименьший вес ребра в этом пути определяет объем дополнительного потока, который можно через него пропустить.

Для иллюстрации этой идеи на рис. 8.12, *а* изображен максимальный поток от $s$ к $t$ в графе $G$ и на рис. 8.12, *б* — связный граф остаточного потока $R(G, f)$. Минимальный разрез потока $s$–$t$ обозначен пунктирной линией возле вершины $t$. Максимальный поток от $s$ к $t$ в графе $G$ равен 7. Этот поток определяется двумя путями в графе остаточного потока $R(G, f)$, направленными от вершины $t$ к вершине $s$ и пропускной способностью 2 и 5 соответственно. Эти потоки полностью используют пропускную способность двух ребер, входящих в вершину $t$, вследствие чего не остается путей, способных что-либо

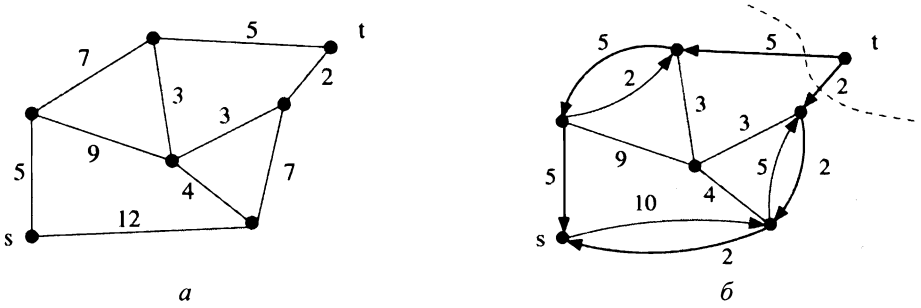



**Рис. 8.12.** *а* — максимальный поток от $s$ к $t$ в графе $G$;
*б* — связный граф остаточного потока $R(G, f)$ и минимальный разрез $s$–$t$ (пунктирная линия рядом с $t$). Неориентированные ребра в $R(G, f)$ имеют нулевой поток, поэтому они обладают остаточной пропускной способностью в обоих направлениях

добавить. Таким образом, поток является оптимальным. Множество ребер, удаление которых отделяет вершину $s$ от вершины $t$ (как два ребра, входящие в вершину $t$ на рис. 8.12, *б*), называется ($s$–$t$)-*разрезом* (cut). Очевидно, что никакой поток из вершины $s$ в вершину $t$ не может превзойти вес такого минимального разреза. С другой стороны, всегда возможен поток, равный минимальному разрезу.

> ***Подведение итогов***
>
> Максимальный поток из $s$ в $t$ всегда равен минимальному весу ($s$–$t$)-разреза. Таким образом, алгоритмы для работы с потоками можно применять для решения общих задач связности ребер и вершин графов.

### Реализация

В этой книге мы не можем представить теорию потоков сети в полном объеме. Но возможность показать, как определять увеличивающие пути и вычислять оптимальный поток, мы не упустим. Для каждого ребра в графе остаточного потока необходимо отслеживать как текущий объем потока, проходящего через него, так и его *остаточную* пропускную способность. Соответственно, необходимо модифицировать структуру ребра, чтобы учесть дополнительные поля (листинг 8.13).

**Листинг 8.13. Модифицированная структура ребра**

```
typedef struct {
   int v;                  /* Соседняя вершина */
   int capacity;           /* Пропускная способность ребра */
   int flow;               /* Поток через ребро */
   int residual;           /* Остаточная пропускная способность ребра */
   struct edgenode *next; /* следующее ребро в списке */
} edgenode;
```

Используя обход в ширину, мы выполняем поиск путей от истока к стоку, которые увеличивают общий поток, и добавляем обнаруженные пути, увеличивая общий поток. Когда все *увеличивающие* пути обнаружены и добавлены в поток, процедура завершается, возвращая оптимальный поток (листинг 8.14).

**Листинг 8.14. Процедура поиска оптимального потока**

```
void netflow(flow_graph *g, int source, int sink) {
    int volume; /* Вес увеличивающего пути */

    add_residual_edges(g);

    initialize_search(g);
    bfs(g, source);

    volume = path_volume(g, source, sink);

    while (volume > 0) {
        augment_path(g, source, sink, volume);
```

```
        initialize_search(g);
        bfs(g, source);
        volume = path_volume(g, source, sink);
    }
}
```

Увеличивающий путь от истока к стоку повышает объем потока, и этот путь можно найти посредством обхода в ширину. Рассматриваются только такие ребра, которые имеют остаточную пропускную способность, или, иными словами, положительный остаточный поток. При обходе в ширину насыщенные и ненасыщенные ребра различаются с помощью процедуры, возвращающей булево значение (листинг 8.15).

**Листинг 8.15. Процедура для различения насыщенных и ненасыщенных ребер**

```
bool valid_edge(edgenode *e) {
    return (e->residual > 0);
}
```

При увеличении пути максимально возможный объем потока перемещается из ребер с остаточной пропускной способностью в положительный поток. Этот объем ограничен ребром пути с наименьшей пропускной способностью, точно так же, как и объем пропускаемой по водопроводу воды ограничен наиболее узкой трубой. Соответствующий код приведен в листинге 8.16.

**Листинг 8.16. Добавление увеличивающих путей в поток**

```
int path_volume(flow_graph *g, int start, int end) {
    edgenode *e;                    /* Рассматриваемое ребро */

    if (parent[end] == -1) {
        return(0);
    }

    e = find_edge(g, parent[end], end);

    if (start == parent[end]) {
        return(e->residual);
    } else {
        return(min(path_volume(g, start, parent[end]), e->residual));
    }
}
```

Вспомним, что при обходе графа в ширину «открыватель» каждой вершины сохраняется в массиве `parent`, что позволяет восстановить кратчайший путь обратно к корню с любой вершины. Ребрами этого дерева являются пары вершин, а не собственно ребра в структуре данных графа, по которому был выполнен обход. Вызов функции `find_edge(g,x,y)` возвращает указатель на запись, кодирующую ребро $(x, y)$ в графе $g$, что необходимо для того, чтобы получить его остаточную пропускную способность.

Функция `find_edge()` может найти этот указатель, просканировав список смежностей вершины $x$ (`g->edges[x]`), или, что даже лучше, взять его из соответствующей таблицы поиска.

Направление дополнительного объема потока по ориентированному ребру $(i, j)$ уменьшает остаточную пропускную способность этого ребра, но *увеличивает* пропускную способность ребра $(j, i)$. Таким образом, действие увеличения пути требует модифицирования как прямых, так и обратных ребер для каждого звена пути. Соответствующий код приведен в листинге 8.17.

**Листинг 8.17. Модификация ребер**

```
void augment_path(flow_graph *g, int start, int end, int volume) {
    edgenode *e;                    /* Рассматриваемое ребро */

    if (start == end) {
        return;
    }

    e = find_edge(g, parent[end], end);
    e->flow += volume;
    e->residual -= volume;

    e = find_edge(g, end, parent[end]);
    e->residual += volume;

    augment_path(g, start, parent[end], volume);
}
```

Для инициализации графа потока требуется создать ориентированные ребра $(i, j)$ и $(j, i)$ для каждого ребра сети $e = (i, j)$. Все начальные потоки инициализируются в 0. Первоначальный остаточный поток ребра $(i, j)$ устанавливается равным пропускной способности ребра сети $e$, а первоначальный остаточный поток ребра $(j, i)$ устанавливается равным нулю.

### Анализ

Алгоритм поиска увеличивающих путей в конечном счете приводит к оптимальному решению. Но каждый новый увеличивающий путь может добавлять только незначительный объем к общему потоку, поэтому теоретически время схождения алгоритма может быть сколь угодно большим.

Однако Эдмондс (Edmonds) и Карп (Karp) в книге [EK72] доказали, что, постоянно выбирая *кратчайший* невзвешенный увеличивающий путь, можно гарантировать, что добавление $O(n^3)$ увеличивающих путей будет достаточным для получения оптимального потока. В действительности, наша реализация поиска оптимального потока и является алгоритмом Эдмондса — Карпа, поскольку для поиска следующего увеличивающего пути используется обход в ширину, начинающийся с истока.

# 8.6. Произвольный минимальный разрез

Хитроумные рандомизированные алгоритмы были разработаны для решения многих различных задач. К этому момнту мы рассмотрели рандомизированные алгоритмы для сортировки (быстрая сортировка), поиска (хеширование), сопоставления строк (алгоритм Рабина — Карпа) и решения теоретико-числовых задач (проверка чисел на простоту). В этом разделе мы добавим к этому списку графовые алгоритмы.

Задача нахождения минимального разреза заключается в разбиении вершин графа $G$ на множества $V_1$ и $V_2$ таким образом, чтобы минимизировать количество ребер $(x, y)$, соединяющих эти два множества, при условии, что $x \in V_1$ и $y \in V_2$. Такая задача часто возникает при анализе надежности сетей — например, при определении наименьшего множества ребер, удаление которого нарушит связность графа. Задача нахождения минимального разреза рассматривается подробно в *разд. 18.8*. Кардинальность минимального разреза примера графа в этом рассмотрении равна 2, а на рис. 8.13, *слева* показан граф, связность которого можно нарушить, удалив только одно ребро.

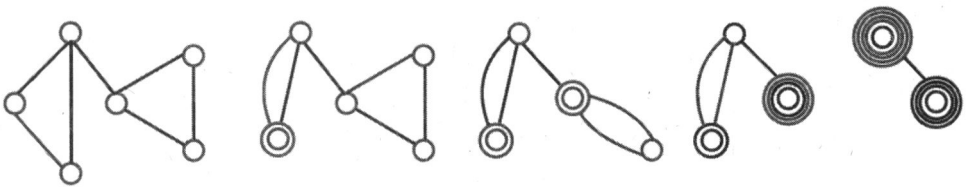


**Рис. 8.13.** Если нам повезет, последовательные стягивания ребер графа не увеличат размер минимального разреза

Предположим, что размер минимального разреза $C$ в графе $G$ равен $k$, — т. е. чтобы сделать граф $G$ несвязным, из него нужно удалить $k$ ребер. При этом каждая вершина $v$ должна быть соединена как минимум с $k$ других вершин, т. к. в противном случае будет существовать разрез меньшего размера, отсоединяющий вершину $v$ от остальной части графа. Это подразумевает наличие в графе $G$ как минимум $kn/2$ ребер, где $n$ представляет количество вершин, поскольку каждое ребро повышает на единицу степень ровно двух вершин.

Операция *стягивания* над ребром $(x, y)$ сворачивает вершины $x$ и $y$ в одну объединенную вершину — назовем ее $xy$. Любое ребро в виде $(x, z)$ или $(y, z)$ заменяется ребром $(xy, z)$. В результате стягивания ребра количество вершин уменьшается на единицу. При этом количество ребер остается прежним, хотя ребро $(x, y)$ заменяется петлей $(xy, xy)$, и создаются две копии ребра $(xy, z)$, если до стягивания как ребро $(x, z)$, так и ребро $(y, z)$ были в графе $G$.

Но что происходит с размером минимального разреза после стягивания ребра $(x, y)$ графа $G$? Каждое стягивание уменьшает пространство возможных разбиений $V_1$ и $V_2$, поскольку для новой вершины $xy$ никак нельзя выполнить разбиение. Здесь важно отметить тот факт, что размер минимального разреза не меняется, если только не стягивается одна из $k$ вершин оптимального разреза. Стягивание одного из ребер разреза может увеличить размер минимального разреза получившегося графа, т. к. наилучшего разбиения больше не будет.

Это подсказывает следующий рандомизированный алгоритм. Выбираем случайное ребро графа $G$ и стягиваем его. Повторяем эту операцию $(n - 2)$ раза до тех пор, пока не получим двухвершинный граф с несколькими параллельными ребрами между этими вершинами. Эти ребра описывают разрез в графе $G$, хотя такой разрез может быть и не самого малого возможного размера. Всю эту процедуру можно было бы повторить $r$ раз и возвратить наименьший полученный разрез в качестве предлагаемого минимального разреза. Время исполнения такой последовательности стягиваний, реализованной должным образом, может составлять $O(nm)$ для любого графа. То есть мы получаем алгоритм Монте-Карло с временем исполнения $O(rmn)$, но не гарантирующий оптимального решения.

Каковы же будут шансы успешного исполнения любой такой итерации? Для этого рассмотрим исходный граф. Стягивание произвольного ребра $e$ сохраняет минимальный разрез $C$ при условии, что это ребро не является одним из ребер $k$ разреза. Так как граф $G$ содержит как минимум $kn/2$ ребер, вероятность $p_i$ успешного стягивания $i$-го ребра будет следующей:

$$p_i \geq 1 - \frac{k}{k(n-i+1)/2} = 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}.$$

Таким образом, для большого графа шансы успешного стягивания всех ребер, за исключением нескольких последних, сильно в нашу пользу. Чтобы для конкретного исполнения алгоритма получить минимальный разрез $C$, нужно, чтобы каждое из $n - 2$ стягиваний было успешным. Вероятность этого будет следующей:

$$\prod_{i=1}^{n-2} p_i = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)}.$$

Члены произведения прекрасно сокращаются, оставляя значение вероятности успеха, равное $\Theta(1/n^2)$. Это не очень большая вероятность, но если исполнить алгоритм $r = n^2 \log n$ раз, то получить по крайней мере один минимальный разрез будет крайне вероятно.

#### Подведение итогов

Ключевой аспект реализации любого рандомизированного алгоритма — организация ситуации, в которой можно установить границы вероятности успешного результата. Соответствующий анализ может быть сопряжен со сложностями, но получаемые алгоритмы часто довольно простые, наподобие рассмотренных в этом разделе. В конце концов, сложные рандомизированные алгоритмы, вероятно, станут слишком трудными для анализа.

## 8.7. Разрабатывайте не алгоритмы, а графы

Правильное моделирование является ключом к эффективному использованию алгоритмов для работы с графами. Мы определили несколько свойств графов и разработали эффективные алгоритмы для их вычисления. В каталоге задач (см. *часть II* книги) представлено около двух десятков разных задач по графам — в основном, в *главах 18* и *19*. Эти классические задачи по графам составляют основу для моделирования большинства приложений.

Но секрет успешного решения задач на графах заключается в умении разрабатывать не алгоритмы для работы с графами, а сами графы. Мы уже видели несколько примеров этой идеи. В частности:

- *максимальное* остовное дерево можно найти, изменив веса ребер исходного графа $G$ на отрицательные и применив на получившемся графе $G'$ алгоритм для построения *минимального* остовного дерева. Остовное дерево графа $G'$ с максимальным по модулю общим весом будет максимальным остовным деревом графа $G$;
- для решения задачи паросочетания в двудольном графе мы создали специальный граф потоков в сети, в котором максимальный поток соответствует паросочетанию, содержащему наибольшее количество ребер.

Рассматриваемые далее примеры дополнительно демонстрируют мощь правильного моделирования. Каждый из этих примеров возник в реальном приложении и каждый можно смоделировать в виде задачи на графах. Некоторые примеры достаточно сложны, но они иллюстрируют универсальность графов в представлении взаимоотношений. Прочитав задачу, не спешите заглядывать в раздел с решением, а попытайтесь создать для нее соответствующее графовое представление.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Нить Ариадны

**ЗАДАЧА**. Требуется создать алгоритм для разработки естественных маршрутов, по которым персонажи видеоигр могут проходить через помещения, наполненные разными препятствиями.

**РЕШЕНИЕ.** Предположительно, искомый маршрут должен соответствовать маршруту, который бы выбрало разумное существо. А поскольку разумные существа склонны к лени и пути наименьшего сопротивления, они выберут самый короткий маршрут. Соответственно, эту задачу нужно моделировать как задачу поиска кратчайшего пути.

Но что у нас будет служить графом? Один из подходов может заключаться в наложении решетки на комнату. Для каждого узла решетки, свободного от предметов, создаем вершину для размещения игрового персонажа. Между любыми двумя близлежащими вершинами создаем ребро, взвешенное пропорционально расстоянию между ними. Хотя для поиска кратчайшего пути существуют прямые геометрические методы (см. *разд. 18.4*), эту задачу легче смоделировать дискретно в виде графа. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Упорядочивание последовательности

**ЗАДАЧА**. Проект секвенирования ДНК имеет на выходе экспериментальные данные, состоящие из небольших фрагментов. Для каждого фрагмента $f$ известно, что некоторые фрагменты обязательно расположены слева от него, а другие — справа. Нужно найти непротиворечивое упорядочивание фрагментов слева направо, которое удовлетворяет всем требованиям.

**РЕШЕНИЕ**. Создаем ориентированный граф, в котором каждый фрагмент представлен вершиной. Вставляем ориентированное ребро $(l, f)$ от любого фрагмента $l$, который должен быть слева от фрагмента $f$, и ориентированное ребро $(f, r)$. к любому фрагменту $r$, который должен быть справа от фрагмента $f$. Нам нужно упорядочить вершины та-

ким образом, чтобы все ребра были направлены слева направо. Это будет *топологическая сортировка* получившегося бесконтурного ориентированного графа. Граф обязательно должен быть бесконтурным, поскольку при наличии контуров (т. е. замкнутых маршрутов) непротиворечивое упорядочивание невозможно. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Размещение прямоугольников по корзинам

**ЗАДАЧА**. Произвольное множество прямоугольников в плоскости нужно распределить по минимальному количеству корзин таким образом, чтобы ни один из прямоугольников в любой корзине не пересекался с другими. Иными словами, в одной и той же корзине не может быть двух прямоугольников с перекрывающимися областями.

**РЕШЕНИЕ.** Создаем граф, в котором каждая вершина представляет прямоугольник, а перекрывающиеся прямоугольники соединяются ребром. Каждая корзина соответствует *независимому множеству* (см. *разд. 19.2*) прямоугольников, поэтому ни один из них не накладывается на другой. *Раскраской вершин* графа (см. *разд. 19.7*) называется разбиение вершин на независимые множества, поэтому для нашей задачи требуется минимизировать количество цветов. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Конфликт имен файлов

**ЗАДАЧА**. При переносе кода из операционной системы UNIX в DOS нужно сократить имена нескольких сотен файлов до самое большее 8 символов. Просто использовать первые восемь символов имени файла нельзя, поскольку `имя_файла1` и `имя_файла2` будут преобразованы в одно и то же имя. Как рационально сократить имена файлов и при этом избежать конфликтов между получившимися именами?

**РЕШЕНИЕ**. Создайте двудольный граф, в котором вершины соответствуют каждому первоначальному имени файла $f_i$ для $1 \le i \le n$, а также набор приемлемых сокращений для каждого имени $f_{i1}$, ..., $f_{ik}$. Соедините ребром каждое первоначальное имя и его сокращенный вариант. Теперь нужно найти набор из $n$ ребер, не имеющих общих вершин, соотнеся, таким образом, первоначальное имя файла с приемлемым индивидуализированным сокращением. Задача *паросочетаний в двудольном графе* как раз и является типом задачи поиска независимого множества ребер в графе. ■

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Разделение текста

**ЗАДАЧА**. Найти способ для разделения строчек текста в разрабатываемой системе распознавания текста. Хотя между печатными строчками текста имеется свободный промежуток, но по разным причинам — таким как помехи или перекос страницы — этот интервал трудно выделить. Каким образом можно решить эту задачу?

**РЕШЕНИЕ**. Примем следующее определение графа. Каждый пиксел изображения представляется вершиной графа, а соседние пикселы соединяются ребром. Вес этого ребра должен быть пропорционален степени черного цвета в пикселах. В таком графе интервал между двумя печатными строчками будет путем, направленным от левого края страницы к правому. Нам нужно найти относительно прямой путь, максимально

избегающий черных точек. Предполагается, что кратчайший путь в графе пикселов будет с большой вероятностью правильным разделителем. ■

#### Подведение итогов

Разработать действительно новый алгоритм работы с графами очень нелегко, поэтому не тратьте на это время. Вместо этого для моделирования стоящей перед вами задачи старайтесь разработать графы, которые позволяют применить классические алгоритмы.

## Замечания к главе

Представление задачи в форме потоков в сети является эффективным алгоритмическим инструментом, но для того чтобы понять, можно ли решить ту или иную задачу с помощью этого инструмента, требуется опыт. Для более подробного изучения этой темы рекомендуется ознакомиться с книгами [CC97] и [Wil19].

Метод увеличивающих путей описан Фордом (Ford) и Фулкерсоном (Fulkerson) в книге [FF62]. Эдмондс (Edmonds) и Карп (Karp) в книге [EK72] доказали, что постоянно выбирая кратчайший невзвешенный увеличивающий путь, можно гарантировать, что добавление $O(n^3)$ увеличивающих путей будет достаточным для получения оптимального потока.

Система распознавания текста, вводимого с помощью кнопочного номеронабирателя телефона, которая рассматривалась ранее в этой главе, описывается более подробно в книге [RS96].

## 8.8. Упражнения

### Алгоритмы для эмуляции графов

1. *[3]* Для графов из задачи 1 *главы* 7:

   a) Нарисуйте остовный лес, получаемый после каждой итерации основного цикла алгоритма Крускала.

   b) Нарисуйте остовный лес, получаемый после каждой итерации основного цикла алгоритма Прима.

   c) Найдите остовное дерево с кратчайшим путем и с корнем в вершине $A$.

   d) Вычислите максимальный поток от вершины $A$ к вершине $H$.

### Минимальные остовные деревья

2. *[3]* Будет ли путь между двумя вершинами в минимальном остовном дереве обязательно самим коротким путем между этими двумя вершинами в полном графе? Если да, предоставьте доказательство, если нет, приведите контрпример.
3. *[3]* Допустим, что все ребра графа имеют разный вес, т. е. нет ни одной пары ребер с одинаковым весом. Будет ли путь между двумя вершинами в минимальном остовном дереве обязательно самым коротким путем между этими двумя вершинами в полном графе? Если да, предоставьте доказательство, если нет, приведите контрпример.

4. *[3]* Могут ли алгоритмы Прима и Крускала выдавать разные минимальные остовные деревья? Аргументируйте свой ответ.

5. *[3]* Могут ли алгоритмы Прима и Крускала работать с графами, содержащими ребра с отрицательным весом? Аргументируйте свой ответ.

6. *[3]*

   a) Предположим, что все ребра графа имеют разный вес, т. е. нет ни одной пары ребер с одинаковым весом. Имеет ли этот граф единственное *минимальное остовное дерево*? Если да, то предоставьте доказательство, если нет, приведите контрпример.

   b) Опять предположим, что все ребра графа имеют разный вес, т. е. нет ни одной пары ребер с одинаковым весом. Имеет ли этот граф единственное *кратчайшее остовное дерево*? Если да, предоставьте доказательство, если нет, приведите контрпример.

7. *[5]* Дано минимальное остовное дерево $T$ графа $G$ (содержащего $n$ вершин и $m$ ребер). К этому графу мы добавим новое ребро $e = (u, v)$ с весом $w$. Разработайте эффективный алгоритм для построения минимального остовного дерева графа $G + e$. Чтобы решение было засчитано полностью, алгоритм должен иметь время исполнения $O(n)$.

8. *[5]*

   a) Дано минимальное остовное дерево $T$ взвешенного графа $G$. Создайте новый граф $G'$, увеличив вес каждого ребра графа $G$ на $k$. Создают ли ребра минимального остовного дерева $T$ графа $G$ минимальное остовное дерево графа $G'$? Если да, предоставьте доказательство, если нет, приведите контрпример.

   b) Пусть $P = \{s, \ldots, t\}$ описывает кратчайший взвешенный путь между вершинами $s$ и $t$ взвешенного графа $G$. Создайте новый граф $G'$, увеличив вес каждого ребра графа $G$ на $k$. Описывает ли $P$ кратчайший путь от вершины $s$ к вершине $t$ в графе $G'$? Если да, предоставьте доказательство, если нет, приведите контрпример.

9. *[5]* Разработайте и выполните анализ алгоритма, который во взвешенном графе $G$ находит наименьшее изменение в стоимости ребра, не являющегося ребром минимального остовного дерева, вследствие которого изменится минимальное остовное дерево графа. Алгоритм должен быть корректным и иметь полиномиальное время исполнения.

10. *[4]* Рассмотрим задачу поиска взвешенного связного подмножества ребер $T$ с минимальным весом во взвешенном связном графе $G$. Вес $T$ состоит из суммы весов всех его ребер.

    a) Почему эта задача не является задачей поиска минимального остовного дерева? Подсказка: вспомните о ребрах с отрицательным весом.

    b) Разработайте эффективный алгоритм поиска связного подмножества $T$ с минимальным весом.

11. *[5]* Пусть T = $(V, E')$ — минимальное остовное дерево графа $G = (V, E)$ с положительными весами ребер. Предположим, что вес определенного ребра $e \in E$ изменен с $w(e)$ на $\hat{w}(e)$. Нам нужно обновить минимальное остовное дерево $T$, чтобы обновить это изменение, но при этом не вычисляя заново все дерево. Для каждого из четырех следующих случаев разработайте алгоритм с линейным временем исполнения для обновления дерева:

    *a)* $e \notin E'$ и $\hat{w}(e) > w(e)$;

    b) $e \notin E'$ и $\hat{w}(e) < w(e)$;

c) $e \in E'$ и $\hat{w}(e) < w(e)$;

d) $e \in E'$ и $\hat{w}(e) > w(e)$.

12. *[4]* Дан неориентированный граф $G = (V, E)$. Множество ребер $F \subseteq E$ называется *разрывающим множеством ребер* (feedback-edge set), если каждый контур в графе $G$ имеет по крайней мере одно ребро в $F$.

    a) Дан невзвешенный граф $G$. Разработайте эффективный алгоритм поиска разрывающего множества ребер минимального размера.

    b) Дан взвешенный неориентированный граф $G$ с ребрами положительного веса. Разработайте эффективный алгоритм поиска разрывающего множества ребер минимального веса.

## Поиск-объединение

13. *[5]* Разработайте эффективную структуру данных для выполнения следующих операций на взвешенных ориентированных графах:

    a) Слияние двух указанных компонентов.

    b) Поиск компонента, содержащего указанную вершину $v$.

    c) Возвращение минимального ребра из указанного компонента.

14. *[5]* Разработайте структуру данных, которая позволяет выполнение такой последовательности из $m$ операций *объединение* и *поиск* на универсальном множестве из $n$ элементов, что вначале выполняются все операции объединения, а за ними — все операции поиска. Последовательность операций должна выполняться за время $O(m + n)$.

## Поиск кратчайшего пути

15. *[3]* В задаче *о кратчайшем пути в пункт назначения* (single-destination shortest path) требуется найти кратчайший путь из каждой вершины графа в указанную вершину. Разработайте эффективный алгоритм для решения этой задачи.

16. *[3]* Дано: неориентированный взвешенный граф $G = (V, E)$ и его остовное дерево $T$ с кратчайшим путем и с корнем в вершине $v$. Если увеличить вес всех ребер графа $G$ на $k$, останется ли $T$ кратчайшим остовным деревом с корнем в вершине $v$?

17. *[3]*

    a) Приведите пример взвешенного связного графа $G = (V, E)$ и вершины $v$, для которых минимальное остовное дерево и кратчайшее остовное дерево с корнем в вершине $v$ являются одинаковыми.

    b) Приведите пример взвешенного связного ориентированного графа $G = (V, E)$ и вершины $v$, для которых минимальное остовное дерево существенно отличается от минимального остовного дерева с корнем в вершине $v$.

    c) Могут ли эти два типа остовных деревьев быть полностью непересекающимися?

18. *[3]* Докажите, что следующие утверждения верны, в противном случае приведите соответствующий контрпример.

    a) Будет ли путь между двумя вершинами в минимальном остовном дереве неориентированного графа кратчайшим (имеющим минимальный вес) путем?

b) Допустим, что граф имеет единственное минимальное остовное дерево. Будет ли путь между двумя вершинами в минимальном остовном дереве неориентированного графа кратчайшим (имеющим минимальный вес)?

19. *[3]* Разработайте эффективный алгоритм для нахождения кратчайшего пути от вершины $x$ к вершине $y$ в неориентированном взвешенном графе $G = (V, E)$ с ребрами положительного веса, при этом полученный путь должен проходить через определенную вершину $z$.

20. *[5]* В некоторых задачах на графах вес может присваиваться не ребрам (или не только ребрам), а вершинам. Пусть $C_v$ означает вес вершины $v$, а $C(x, y)$ — вес ребра $(x, y)$. Требуется найти самый дешевый путь между вершинами $a$ и $b$ графа $G$. Стоимость пути определяется как сумма весов ребер и вершин, через которые проходит путь.

    a) Все ребра графа имеют нулевой вес, а отсутствующие ребра обозначаются бесконечным весом ($\infty$). Вес каждой вершины $C_v = 1$ ($1 \le v \le n$). Разработайте *эффективный* алгоритм поиска самого дешевого пути от вершины $a$ к вершине $b$. Укажите временную сложность этого алгоритма.

    a) Вершины имеют разный вес (но обязательно положительный), а вес ребер остается прежним. Разработайте *эффективный* алгоритм поиска самого дешевого пути от вершины $a$ к вершине $b$. Укажите временную сложность этого алгоритма.

    b) Вершины и ребра имеют разный вес (но обязательно положительный). Разработайте *эффективный* алгоритм поиска самого дешевого пути от вершины $a$ к вершине $b$. Укажите временную сложность этого алгоритма.

21. *[5]* Разработайте алгоритм с временем исполнения $O(n^3)$, который возвращает длину самого короткого пути в ориентированном графе $G$ с $n$ вершинами и ребрами положительной длины. В случае ациклического графа длина пути будет равна $\infty$.

22. *[5]* Сеть автомобильных дорог представлена взвешенным графом $G$, ребра которого соответствуют дорогам, а вершины — пересечениям дорог. Для каждой дороги указан максимальный разрешенный вес транспортных средств для проезда по ней. Разработайте эффективный алгоритм для вычисления максимального возможного веса транспортных средств для законного проезда от точки $s$ к точке $t$. Какова временная сложность вашего алгоритма?

23. *[5]* Дан ориентированный граф $G$, возможно, содержащий ребра отрицательного веса, в котором кратчайший путь между любыми двумя вершинами гарантированно состоит из самое большее $k$ ребер. Разработайте алгоритм для нахождения кратчайшего пути между двумя вершинами $u$ и $v$ за время $O(k \cdot (n + m))$.

24. *[5]* Можно ли решить задачу поиска *самого длинного* пути из заданного пункта выхода (single-source *longest* path) модифицированным алгоритмом Дейкстры, в котором `minimum` изменено на `maximum`? Если да, предоставьте доказательство, если нет, приведите контрпример.

25. *[5]* Дан взвешенный бесконтурный ориентированный граф $G = (V, E)$, в котором, возможно, имеются ребра с отрицательным весом. Разработайте алгоритм с линейным временем исполнения для решения задачи поиска кратчайшего пути из заданной начальной вершины $v$.

26. *[5]* Дан взвешенный ориентированный граф $G = (V, E)$, в котором все веса положительные. Пусть $v$ и $w$ — вершины графа $G$, $k$ — целое число ($k \le |V|$). Разработайте алгоритм

поиска кратчайшего пути от вершины $v$ к вершине $w$, содержащего ровно $k$ ребер. Путь не обязательно должен быть простым.

27. *[5] Арбитражем* называется использование разницы в курсах обмена валют для получения прибыли. Например, в течение короткого периода времени за 1 доллар США можно купить 0,75 фунта стерлингов, за 1 фунт стерлингов — 2 австралийских доллара, а за 1 австралийский доллар — 0,50 доллара США. То есть осуществление такой сделки принесет прибыль, равную $0{,}75 \times 2 \times 0{,}7 = 1{,}05$ доллара США, или 5%. Имеется $n$ валют $c_1$, ..., $c_n$ и таблица курсов валют $R$ размером $n \times n$, в которой указывается, что за одну единицу валюты $c_i$ можно купить $R[i, j]$ единиц валюты $c_j$. Разработайте и выполните анализ алгоритма для определения максимального значения:

$$R[c_1, c_{i1}] \cdot R[c_{i1}, c_{i2}] \cdot \cdot \cdot R[c_{ik-1}, c_{ik}] \cdot R[c_{ik}, c_1]$$

Подсказка: ищите кратчайший путь между всеми парами вершин.

## Потоки в сети и паросочетание

28. *[3] Паросочетанием* в графе называется набор непересекающихся ребер — т. е. ребер, которые не имеют общих вершин. Разработайте алгоритм поиска максимального паросочетания в дереве.

29. *[5] Реберным покрытием* (edge cover) неориентированного графа $G = (V, E)$ называется набор ребер, для которого в каждую вершину графа входит по крайней мере одно ребро из этого набора. Разработайте эффективный алгоритм на основе паросочетаний для поиска реберного покрытия максимального размера графа $G$.

## LeetCode

1. **https://leetcode.com/problems/cheapest-flights-within-k-stops/**
2. **https://leetcode.com/problems/network-delay-time/**
3. **https://leetcode.com/problems/find-the-city-with-the-smallest-numberof-neighbors-at-a-threshold-distance/**

## HackerRank

1. **https://www.hackerrank.com/challenges/kruskalmstrsub/**
2. **https://www.hackerrank.com/challenges/jack-goes-to-rapture/**
3. **https://www.hackerrank.com/challenges/tree-pruning/**

## Задачи по программированию

Эти задачи доступны на сайте **http://onlinejudge.org**:

1. «Freckles», глава 10, задача 10034.
2. «Necklace», глава 10, задача 10054.
3. «Railroads», глава 10, задача 10039.
4. «Tourist Guide», глава 10, задача 10199.
5. «The Grand Dinner», глава 10, задача 10249.

ГЛАВА 9

# Комбинаторный поиск

Задачи на удивление большого объема можно решить, используя методы исчерпывающего поиска, хотя и с большими вычислительными затратами. Но для некоторых приложений эти затраты могут быть оправданы. Например, правильность работы электрической схемы можно доказать, проверив все возможные входные сигналы и удостоверившись в правильности выходного сигнала. Доказанная абсолютная правильность устройства будет очень желаемым свойством. А вот доказанная правильность только для всех перебранных входных сигналов имеет намного меньшую ценность.

Частота тактового генератора современных компьютеров достигает нескольких *гигагерц*, а это означает, что они могут исполнять несколько миллиардов базовых инструкций в секунду. Так как осуществление какой-либо представляющей интерес операции более высокого уровня требует нескольких сот базовых инструкций, то можно ожидать, что за секунду мы можем просмотреть несколько миллионов элементов.

Но важно иметь представление, насколько велик один миллион. Один миллион перестановок означает все возможные упорядочивания приблизительно 10 объектов, но не более. Один миллион подмножеств означает все возможные комбинации около 20 элементов, но не более. Решение задач существенно большего размера требует тщательного сокращения пространства поиска, чтобы обработке подвергались только действительно имеющие важность элементы.

В этой главе представляется метод перебора с возвратом, применяющийся для перечисления всех возможных решений комбинаторной алгоритмической задачи. Я также покажу несколько интересных методов отсечения тупиковых решений для ускорения работы реальных поисковых приложений. Решение же задач, слишком больших для применения метода полного перебора всех комбинаций, осуществляется на основе рассмотреных в *главе 12* эвристических алгоритмов — таких как имитация отжига. Подобные эвристические алгоритмы являются важными инструментами в наборе любого практикующего алгоритмиста.

## 9.1. Перебор с возвратом

Перебор с возвратом (backtracking) позволяет систематически исследовать все возможные конфигурации области поиска. Эти конфигурации могут представлять все возможные расположения объектов (т. е. перестановки) или все возможные наборы объектов (т. е. подмножества). В других распространенных ситуациях может потребоваться выполнить перечисление всех деревьев графа, всех путей между двумя вершинами или всех возможных способов группирования вершин по цветам.

Общий момент в этих задачах — то, что каждую возможную конфигурацию нужно сгенерировать *только лишь* один раз. Запрет как на повторение, так и на пропуск конфигураций означает, что нам нужно определить четкий порядок их генерирования. Мы будем моделировать наше решение в виде вектора $a = (a_1, a_2, ..., a_n)$, в котором каждый элемент $a_i$ выбирается из конечного упорядоченного множества $S_i$. Такой вектор может представлять конфигурацию, в которой $a_i$ содержит $i$-й элемент перестановки. Или же, возможно, это булев вектор, представляющий заданное подмножество $S$, где $a_i$ истинно тогда и только тогда, когда $i$-й элемент содержится в $S$. Этот же вектор решения может также представлять последовательность ходов в игре или путь в графе, где $a_i$ содержит $i$-й ход игры или ребро графа в соответствующей последовательности.

На каждом этапе алгоритма перебора с возвратом мы пытаемся расширить частичное решение $a = (a_1, a_2, ..., a_k)$, добавляя следующий элемент в конец последовательности. После этого расширения последовательности нам нужно проверить, не содержит ли она полного решения, и если содержит, то мы можем вывести его или добавить в список полных решений. В противном случае нам нужно выяснить, существует ли возможность расширения частичного решения до полного решения.

При переборе с возвратом создается дерево, в котором каждый узел представляет частичное решение. Если узел $y$ создан в результате расширения узла $x$, то эти узлы соединяются ребром. Такое дерево частичных решений предоставляет альтернативный взгляд на перебор с возвратом, поскольку процесс создания решений в точности соответствует процессу обхода в глубину дерева перебора с возвратом. Рассматривая перебор с возвратом как обход в глубину (depth-first search, DFS) неявного графа, мы создаем естественную рекурсивную реализацию базового алгоритма, псевдокод которого показан в листинге 9.1.

**Листинг 9.1. Перебор с возвратом**

```
Backtrack-DFS(a, k)
    if a = (a1, a2, ..., ak) является решением, выводим его.
    else
        k = k + 1
        создаем Sk, множество кандидатов k для a
        while Sk ≠ ∅ do
            ak = элемент в Sk
            Sk = Sk - {ak}
            Backtrack-DFS(a, k)
```

Хотя для перечисления решений можно было бы также применить и обход в ширину (breadth-first search, BFS), обход в глубину намного предпочтительнее, поскольку он занимает меньше места. Текущее состояние поиска полностью представляется путем от корня к текущему узлу обхода в глубину. Требуемое для этого место пропорционально высоте дерева. А при обходе в ширину в очереди сохраняются все узлы текущего уровня, для чего нужно место, пропорциональное ширине дерева поиска. Для большинства представляющих интерес задач ширина дерева возрастает экспоненциально по отношению к его высоте.

### Реализация

Код универсального алгоритма перебора с возвратом приведен в листинге 9.2.

**Листинг 9.2. Реализация алгоритма перебора с возвратом**

```
void backtrack(int a[], int k, data input) {
    int c[MAXCANDIDATES];       /* Кандидаты для следующей позиции */
    int nc;                     /* Количество кандидатов на следующую позицию */
    int i;                      /* Счетчик */

    if (is_a_solution(a, k, input)) {
        process_solution(a, k,input);
    } else {
        k = k + 1;
        construct_candidates(a, k, input, c, &nc);
        for (i = 0; i < nc; i++) {
            a[k] = c[i];
            make_move(a, k, input);
            backtrack(a, k, input);
            unmake_move(a, k, input);

            if (finished) {
                return;             /* Досрочное завершение */
            }
        }
    }
}
```

Метод перебора с возвратом обеспечивает правильность результата, перечисляя все возможные комбинации, а эффективность его обеспечивается тем, что никакое состояние не исследуется более одного раза.

Разберитесь, как рекурсия позволяет создать легкую и элегантную реализацию алгоритма перебора с возвратом. Поскольку при каждом рекурсивном вызове создается новый массив кандидатов `c`, то подмножества еще не рассмотренных кандидатов на расширение решения в каждой позиции не будут пересекаться друг с другом.

Алгоритм содержит пять процедур, специфичных для конкретных приложений:

- `is_a_solution(a,k,input)`.

  Эта булева функция проверяет, составляют ли первые `k` элементов вектора `a` полное решение задачи. Последний аргумент — `input` — позволяет передавать в процедуру общую информацию. Например, с его помощью можно указать значение $n$, представляющее заданный размер решения. Эта информация может быть полезной при создании перестановок или подмножеств из $n$ элементов, но при создании объектов переменного размера, таких как последовательности ходов игры, можно передавать другие данные, более соответствующие ситуации.

- `construct_candidates(a,k,input,c, &nc)`.

  Эта процедура записывает в массив `c` полный набор возможных кандидатов на `k`-ю позицию вектора `a` при заданном содержимом первых `k` – 1 позиций. Количест-

во кандидатов, содержащихся в этом массиве, заносится в переменную nc. Так же, как в предыдущей функции, аргумент input можно использовать для передачи в процедуру вспомогательной информации;

- process_solution(a,k,input).

  Эта процедура выводит, вычисляет, сохраняет или обрабатывает полное решение после его создания;

- make_move(a,k,input) и unmake_move(a,k,input).

  Эти процедуры позволяют модифицировать структуру данных в ответ на последнее перемещение, а также очистить структуру данных, если мы решим отменить это перемещение. При необходимости эту структуру всегда можно воссоздавать с нуля на основе вектора решений a, но этот подход может быть неэффективным, когда с каждым перемещением связаны небольшие инкрементальные изменения, которые можно с легкостью отменить.

  Указанные процедуры функционируют в виде заглушек (т. е. ничего не делают) в вызовах процедуры backtrack() во всех примерах этого раздела, но применяются в программе решения головоломок судоку в *разд. 9.4*.

Для внепланового завершения программы используется глобальный флаг finished, который можно установить в любой прикладной процедуре.

# 9.2. Примеры перебора с возвратом

Чтобы по-настоящему понять принцип работы алгоритма перебора с возвратом, нужно разобраться, как можно создавать такие объекты, как перестановки и подмножества, определяя правильное пространство состояний. Несколько примеров пространств состояний рассматривается в последующих подразделах.

## 9.2.1. Генерирование всех подмножеств

Разработка подходящего пространства состояний для представления комбинаторных объектов начинается с подсчета количества объектов, которые нужно представить. Сколько существует подмножеств множества из $n$ элементов — например, множества целых чисел $\{1, ..., n\}$? Для $n = 1$ существуют два таких подмножества: $\{\}$ и $\{1\}$. Для $n = 2$ существуют четыре подмножества, а для $n = 2$ — восемь. Как можно видеть, количество подмножеств удваивается с каждым новым элементом множества — таким образом, для множества из $n$ элементов существует $2^n$ подмножеств.

Каждое подмножество описывается содержащимися в нем элементами. Чтобы сгенерировать все $2^n$ подмножеств, мы создаем булев массив (вектор) из $n$ ячеек, в котором булево значение $a_i$ указывает, содержит ли это подмножество $i$-й элемент. В схеме нашего общего алгоритма перебора с возвратом $S_k = (true, false)$, в то время как значение $a$ является решением при $k = n$. Теперь мы можем сгенерировать все подмножества, используя простые реализации процедур is_a_solution(), construct_candidates() и process_solution(), приведенные в листинге 9.3.

**Листинг 9.3. Реализация базовых процедур процедуры `backtrack()`**

```
int is_a_solution(int a[], int k, int n) {
    return (k == n);
}

void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    c[0] = true;
    c[1] = false;
    *nc = 2;
}

void process_solution(int a[], int k, int input) {
    int i;                                          /* Счетчик */

    printf("{");
    for (i = 1; i <= k; i++) {
        if (a[i] == true) {
            printf(" %d", i);
        }
    }

    printf(" }\n");
}
```

Самой сложной из этих трех процедур оказывается процедура для вывода каждого подмножества после его создания!

Наконец, при вызове процедуры `backtrack` ей нужно передать соответствующие аргументы. Конкретно это означает: предоставление указателя на пустой вектор решения, установление *k* в ноль для обозначения того, что вектор действительно пустой, и указание количества элементов в универсальном множестве (листинг 9.4).

**Листинг 9.4. Вызов процедуры `backtrack()` для генерирования подмножеств**

```
void generate_subsets(int n)
{
   int a[NMAX];      /* Вектор решений */
   backtrack(a, 0, n);
}
```

В каком порядке будут генерироваться подмножества множества {1, 2, 3}? Это зависит от порядка перемещений, возвращаемых процедурой `construct_candidates`. Так как *true* всегда идет перед *false*, то сначала будут сгенерированы все подмножества для *true*, а пустое подмножество, состоящее из всех *false*, генерируется последним: {123}, {12}, {13}, {1}, {23}, {2}, {3} и {}.

Изучите внимательно приведенный пример (в этом вам поможет рис. ЦВ-9.1, *а*) и убедитесь, что вы понимаете, как работает процедура перебора с возвратом. Задача генерирования подмножеств рассматривается более подробно в *разд. 17.5*.

## 9.2.2. Генерирование всех перестановок

Обязательным предварительным условием генерирования перестановок множества элементов $\{1, ..., n\}$ является подсчет их количества. Для первого элемента перестановки имеется $n$ вариантов. После выбора элемента $a_1$ вторым элементом может быть любой из оставшихся $n-1$ элементов, кроме элемента $a_1$ (т. к. в перестановках повторение элементов не допускается). Последовательное применение этой процедуры дает нам

$$n! = \prod_{i=1}^{n} i$$

разных перестановок.

Этот способ подсчета количества перестановок подсказывает подходящую структуру для их представления. В частности, создаем массив (вектор) $a$ из $n$ ячеек. Набором кандидатов на $i$-е место будут все элементы, которые не вошли в $(i-1)$ элементов частичного решения, что соответствует первым $i-1$ элементам перестановки.

В схеме общего алгоритма перебора с возвратом $S_k = \{1, ..., n\} - \{a_1, ..., a_k\}$, причем значение $a$ является решением, когда $k = n$. Соответствующая процедура `construct_candidates()` представлена в листинге 9.5.

**Листинг 9.5. Процедура `construct_candidates()` для генерирования всех перестановок**

```
void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    int i;                      /* Счетчик */
    bool in_perm[NMAX];         /* Какие элементы в перестановке? */

    for (i = 1; i < NMAX; i++) {
        in_perm[i] = false;

    }
    for (i = 1; i < k; i++) {
        in_perm[a[i]] = true;
    }

    *nc = 0;
    for (i = 1; i <= n; i++) {
        if (!in_perm[i]) {
            c[ *nc ] = i;
            *nc = *nc + 1;
        }
    }
}
```

Узнать, является ли $i$-й элемент кандидатом на $k$-е место в перестановке, можно было бы путем перебора всех элементов $k-1$ массива $a$, чтобы убедиться в отсутствии совпадений. Но гораздо лучший способ отслеживания элементов, находящихся в частичном решении, — организовать структуру данных в форме вектора разрядов (см. *разд. 15.5*), что позволит нам выполнять проверку за постоянное время.

Для завершения программы генерирования перестановок необходимо определить используемые в ней процедуры `process_solution` и `is_a_solution`, а также передать необходимые параметры вызываемой процедуре `backtrack`. И определение процедур, и передача параметров выполняются, по сути, так же, как и для программы генерирования подмножеств (листинг 9.6).

**Листинг 9.6. Процедуры генерирования перестановок**

```
void process_solution(int a[], int k, int input) {
    int i;                          /* Счетчик */

    for (i = 1; i <= k; i++) {
        printf(" %d", a[i]);
    }
    printf("\n");
}

    int is_a_solution(int a[], int k, int n) {
        return (k == n);
}

void generate_permutations(int n) {
    int a[NMAX];                    /* Вектор решений */

    backtrack(a, 0, n);
}
```

В результате упорядочения кандидатов эти перестановки генерируются в *лексикографическом*, или отсортированном, порядке — т. е.: 123, 132, 213, 231, 312 и 321 (рис. ЦВ-9.1, *б*). Задача генерирования перестановок рассматривается более подробно в *разд. 17.4*.

## 9.2.3. Генерирование всех путей в графе

*Простой путь* не содержит более одного вхождения любой вершины. Задача перечисления всех простых путей от вершины $s$ к вершине $t$ графа является более сложной, чем простое перечисление подмножеств или перестановок множества элементов. Не существует явной формулы для определения количества решений в зависимости от количества вершин и ребер, поскольку количество путей зависит от структуры графа.

Входные данные, которые нужно передать процедуре `backtrack` для построения пути, состоят из входного графа $g$, исходной вершины $s$ и конечной вершины $t$. Создание соответствующей структуры данных приведено в листинге 9.7.

**Листинг 9.7. Создание структуры для хранения входных данных для процедуры `backtrack`**

```
typedef struct {
    int s; /* исходная вершина */
```

```
    int t; /* конечная вершина */
    graph g; /* граф для поиска путей */
} paths_data;
```

Начальной точкой любого пути от вершины $s$ к вершине $t$ всегда является вершина $s$ — соответственно, вершина $s$ есть единственный кандидат на первое место и $S_1 = \{s\}$. Возможными кандидатами на второе место в пути являются такие вершины $v$, для которых ребро $(s, v)$ находится в графе, поскольку допустимый путь от вершины к вершине идет по ребрам между ними. Вообще говоря, множество $S_{k+1}$ состоит из набора вершин, смежных с вершиной $a_k$, которые не были использованы в частичном решении $a$. Определение соответствующей процедуры `construct_candidates()` приведено в листинге 9.8.

**Листинг 9.8. Процедура `construct_candidates()` для перечисления всех путей в графе**

```
void construct_candidates(int a[], int k, paths_data *g, int c[],
    int *nc) {
    int i;                      /* Счетчики */
    bool in_sol[NMAX+1];        /* Что уже находится в решении? */
    edgenode *p;                /* Временный указатель */
    int last;                   /* Последняя вершина в текущем пути */

    for (i = 1; i <= g->g.nvertices; i++) {
        in_sol[i] = false;
    }

    for (i = 0; i < k; i++) {
        in_sol[a[i]] = true;
    }

    if (k == 1) {
        c[0] = g->s;            /* Всегда начинаем с вершины s */
        *nc = 1;
    } else {
        *nc = 0;
        last = a[k-1];
        p = g->g.edges[last];
        while (p != NULL) {
            if (!in_sol[ p->y ]) {
                c[*nc] = p->y;
                *nc= *nc + 1;
            }
            p = p->next;
        }
    }
}
```

Условием правильного пути является $a_k = t$. Процедуры для определения решения и его обработки приводятся в листинге 9.9.

**Листинг 9.9. Процедуры для определения решения и его обработки**

```
int is_a_solution(int a[], int k, paths_data *g) {
    return (a[k] == g->t);
}
```

Количество обнаруженных путей можно подсчитать в процедуре `process_solution`, инкрементируя глобальную переменную `solution_count`. Последовательность вершин для каждого пути сохраняется в векторе решения *a* и готова для отображения. Код процедуры `process_solution` приводится в листинге 9.10.

**Листинг 9.10. Процедура для подсчета количества обнаруженных путей**

```
void process_solution(int a[], int k, paths_data *input) {
    int i; /* Счетчик */

    solution_count ++;

    printf("{");
    for (i = 1; i <= k; i++) {
        printf(" %d",a[i]);
    }
    printf(" }\n");
}
```

Вектор решений *a* должен иметь достаточно элементов для представления всех *n* вершин, хотя большинство путей должны быть более короткими. На рис. ЦВ-9.2 показано дерево поиска с перечислением всех путей между определенной парой вершин графа.

# 9.3. Отсечение вариантов поиска

Метод перебора с возвратом обеспечивает правильность результата, перечисляя все возможные комбинации. Правильный алгоритм для определения оптимального маршрута коммивояжера перечисляет все $n!$ перестановок $n$ вершин графа $G$. Для каждой перестановки выполняется проверка, в действительности ли граф $G$ содержит все ребра, указываемые в маршруте, и если содержит, то веса этих ребер суммируются. Решением будет маршрут с минимальным весом.

Но предварительное генерирование всех перестановок для их дальнейшего исследования будет расточительным расходованием ресурсов. Допустим, что наш путь начинается в вершине $v_1$, но пара вершин $(v_1, v_2)$ не является ребром графа $G$. Тогда рассмотрение следующих $(n-2)!$ перестановок, сгенерированных, начиная с ребра $(v_1, v_2)$, окажется бессмысленной тратой времени. Разумнее отказаться от поиска после $(v_1, v_2)$ и продолжить с $(v_1, v_3)$. Ограничение набора следующих элементов таким образом, чтобы остались лишь перемещения, допустимые для текущей частичной конфигурации, позволяет значительно понизить общую сложность поиска.

*Отсечением* (pruning) называется метод отказа от дальнейшего поиска решения в каком-либо направлении, как только можно установить, что это частичное решение не-

возможно расширить до полного. Задача коммивояжера состоит в определении самого дешевого маршрута, который проходит через все вершины. Допустим, что мы нашли маршрут $t$ со стоимостью $C_t$. Потом в процессе продолжения поиска мы получаем частичное решение с суммой стоимости вершин $C_a \geq C_t$. Нужно ли продолжать исследование этого узла? Нет, т. к. любой маршрут с префиксом $a$ будет стоить больше, чем маршрут $t$, и поэтому наверняка не является оптимальным. Отсечение таких бесперспективных частичных маршрутов из дерева поиска на раннем этапе может существенно улучшить время исполнения.

Другим средством уменьшения вариантов возможных решений комбинаторного поиска является применение симметрии. Отсечение частичных решений, эквивалентных рассмотренным ранее, требует умения распознавать симметричные области поиска. Возьмем, например, состояние поиска кратчайшего пути после того, как были рассмотрены все частичные решения, начиная с $v_1$. Есть ли смысл продолжать поиск с частичными решениями, начинающимися с $v_2$? Нет. Любой маршрут, начинающийся и заканчивающийся в вершине $v_2$, можно рассматривать как смещенный маршрут, начинающийся и заканчивающийся в вершине $v_1$, поскольку маршруты поиска кратчайшего пути являются замкнутыми циклами. Таким образом, для $n$ вершин существует только $(n-1)!$ разных маршрутов, а не $n!$. Ограничивая первый элемент маршрута вершиной $v_1$, мы получаем экономию времени порядка $n$, не пропуская при этом никаких представляющих интерес решений. Такие закономерности могут быть далеко не очевидными, но, когда они обнаружены, их можно эксплуатировать.

#### Подведение итогов

Комбинаторный поиск можно использовать совместно с методами отсечения для решения задач оптимизации небольшого размера. Смысл понятия «небольшой» зависит от конкретной задачи, но обычно размер составляет $20 \leq n \leq 100$ элементов.

## 9.4. Судоку

Головоломки судоку очень популярны во всем мире. Многие газеты печатают их в своих дневных тиражах, выпущены целые сборники этих головоломок. О популярности судоку можно судить по тому факту, что авиакомпания British Airways издала приказ, запрещающий бортпроводникам решать их во время взлета и посадки. Я даже заметил, что на моих лекциях по алгоритмам на задних рядах аудитории довольно многие занимаются решением этих головоломок.

Что же представляет собой судоку? В наиболее распространенной форме это квадрат размером 9×9 клеточек, некоторые из них содержат цифры от 1 до 9, а остальные — пустые. Решение головоломки состоит в заполнении пустых клеточек таким образом, чтобы каждая строка, каждый столбец и каждый малый квадрат размером 3×3 клеточки содержали все цифры от 1 до 9, без повторений и без пропусков. Пример головоломки судоку и ее решение показаны на рис. 9.3.

Судоку хорошо поддается решению методом перебора с возвратом. Мы воспользуемся головоломкой, показанной на рис. 9.3, чтобы наглядно проиллюстрировать метод отсечения для комбинаторного поиска. В качестве пространства состояний примем

набор пустых клеточек, каждая из которых будет в конечном итоге заполнена какой-либо цифрой. Кандидатами на заполнение пустой клеточки $(i, j)$ являются целые числа от 1 до 9, которых еще нет в строке $i$, столбце $j$ и в малом квадрате, содержащих клеточку $(i, j)$. Возврат осуществляется, когда больше нет кандидатов на заполнение клеточки.

| | | | | | | | 1 | 2 |
|---|---|---|---|---|---|---|---|---|
| | | | | 3 | 5 | | | |
| | | | 6 | | | | 7 | |
| 7 | | | | | | 3 | | |
| | | | 4 | | | 8 | | |
| 1 | | | | | | | | |
| | | | 1 | 2 | | | | |
| | 8 | | | | | | 4 | |
| | 5 | | | | | 6 | | |

*а*

| 6 | 7 | 3 | 8 | 9 | 4 | 5 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|
| 9 | 1 | 2 | 7 | 3 | 5 | 4 | 8 | 6 |
| 8 | 4 | 5 | 6 | 1 | 2 | 9 | 7 | 3 |
| 7 | 9 | 8 | 2 | 6 | 1 | 3 | 5 | 4 |
| 5 | 2 | 6 | 4 | 7 | 3 | 8 | 9 | 1 |
| 1 | 3 | 4 | 5 | 8 | 9 | 2 | 6 | 7 |
| 4 | 6 | 9 | 1 | 2 | 8 | 7 | 3 | 5 |
| 2 | 8 | 7 | 3 | 5 | 6 | 1 | 4 | 9 |
| 3 | 5 | 1 | 9 | 4 | 7 | 6 | 2 | 8 |

*б*

**Рис. 9.3.** Головоломка судоку высокого уровня сложности (*а*) и ее решение (*б*)

Вектор решения `a`, поддерживаемый процедурой `backtrack`, может содержать в каждой ячейке только одно целое число. Этого достаточно для хранения содержимого клеточки (числа 1–9), но не для хранения ее координат. Поэтому для хранения позиций ходов мы используем отдельный массив `boardtype`. Основные структуры данных для поддержки нашего решения определены в листинге 9.11.

**Листинг 9.11. Определение основных структур данных**

```
#define DIMENSION 9                    /* Доска размером 9*9 */
#define NCELLS DIMENSION*DIMENSION     /* Доска размером 9*9 содержит 81 клеточку */
#define MAXCANDIDATES DIMENSION+1      /* Максимальное количество цифр для клеточки
*/

bool finished = false;

typedef struct {
    int x, y;                           /* Координаты x и y клеточки */
} point;

typedef struct {
    int m[DIMENSION+1][DIMENSION+1];    /* Содержимое доски */
    int freecount;                      /* Количество оставшихся пустых клеточек */
    point move[NCELLS+1];               /* Заполненные клеточки */
} boardtype;
```

На очередном шаге игры мы должны сначала выбрать открытую клеточку, которую хотим заполнить следующей (процедура `next_square`), затем определить цифры,

являющиеся кандидатами на заполнение этой клеточки (процедура possible_values). Эти процедуры (листинг 9.12), по сути, ведут учет ходов, хотя некоторые детали их работы могут значительно повлиять на производительность.

**Листинг 9.12. Генерирование кандидатов на заполнение клеточки**

```
void construct_candidates(int a[], int k, boardtype *board, int c[],
    int i;                          /* Счетчик */
    bool possible[DIMENSION+1];     /* Какие цифры можно использовать для этой
                                                                клеточки */

    next_square(&(board->move[k]), board); /* Выбираем клеточку для заполнения
                                                                следующей  */

    *nc = 0;

    if ((board->move[k].x < 0) && (board->move[k].y < 0)) {
        return;                         /* Ошибка: нет допустимых ходов */
    }

    possible_values(board->move[k], board, possible);
    for (i = 1; i <= DIMENSION; i++) {
        if (possible[i]) {
            c[*nc] = i;
            *nc = *nc + 1;
        }
    }
}
```

Структуры данных игровой доски необходимо обновлять, чтобы отображать заполнение клеточки значением кандидата, а также очищать заполненные клеточки в случае необходимости возврата с отработанной позиции. Для выполнения этих обновлений используются процедуры make_move и unmake_move (листинг 9.13), которые вызываются непосредственно из процедуры backtrack.

**Листинг 9.13. Процедуры make_move и unmake_move**

```
void make_move(int a[], int k, boardtype *board) {
    fill_square(board->move[k], a[k], board);
}

void unmake_move(int a[], int k, boardtype *board) {
    free_square(board->move[k], board);
}
```

Одной из важных задач, выполняемых этими процедурами, является отслеживание количества остающихся на доске пустых клеточек. Решение найдено, когда на доске больше нет пустых клеточек (листинг 9.14). Здесь steps — это глобальная переменная, фиксирующая сложность нашего поиска в табл. 9.1 (см. далее).

**Листинг 9.14. Процедура отслеживания пустых клеточек**

```
bool is_a_solution(int a[], int k, boardtype *board) {
    steps = steps + 1;      /* Подсчитываем количество шагов таблицы результатов */
    return (board->freecount == 0);
}
```

Когда решение найдено, устанавливается глобальный флаг `finished`, что служит сигналом к прекращению поиска и выводу решения. Это можно делать, не опасаясь никаких последствий, поскольку классические головоломки судоку могут иметь только одно решение. Головоломки с расширенной интерпретацией могут иметь громадное количество решений. В самом деле, для пустой головоломки (т. е. без начальных цифр) существует 6 670 903 752 021 072 936 960 решений. Чтобы не просматривать все эти решения, мы и прекращаем поиск (листинг 9.15).

**Листинг 9.15. Завершение поиска и обработка решения**

```
void process_solution(int a[], int k, boardtype *board) {
    finished = true;
    printf("process solution\n");
    print_board(board);
}
```

Эта процедура завершает нашу программу, но остается необходимость в написании процедур, определяющих следующую клеточку для заполнения (`next_square`) и поиск кандидатов на заполнение этой клеточки (`possible_values`). Следующую клеточку для заполнения можно выбрать при помощи одного из следующих двух естественных эвристических алгоритмов:

- *Выбор произвольной клеточки.*

  Выбираем первую попавшуюся пустую клеточку. Нам все равно, какую выбирать, поскольку нет очевидных оснований считать, что один эвристический метод окажется лучше другого.

- *Выбор клеточки с наименьшим количеством кандидатов.*

  При этом подходе мы проверяем количество оставшихся кандидатов на заполнение каждой пустой клеточки $(i, j)$ — т. е. количество цифр, которые еще не используются ни в строке $i$, ни в столбце $j$, ни в малом квадрате, содержащих клеточку $(i, j)$. По результатам этих исследований мы выбираем клеточку с наименьшим количеством кандидатов на ее заполнение.

Хотя оба подхода дают правильные результаты, второй позволяет найти решение намного быстрее. В случае наличия пустых клеточек только с одним кандидатом, которые нельзя заполнить иной цифрой, кроме как этим единственным оставшимся кандидатом, их вполне можно заполнить первыми, поскольку это поможет уменьшить количество возможных кандидатов для заполнения других пустых клеточек. Конечно же, в этом случае выбор каждой следующей клеточки для заполнения будет занимать больше времени, но если головоломка достаточно легкая, то нам, может быть, никогда не придется выполнять возврат.

Если для клеточки с наименьшим количеством кандидатов имеются два кандидата, то вероятность угадать правильный из них с первого раза равна 50%, по сравнению с вероятностью 1/9 угадать правильного кандидата для клеточки без ограничений на количество кандидатов. Уменьшив среднее количество кандидатов для каждой клеточки — например, с трех до двух, мы получим громадное повышение производительности, поскольку это уменьшение дает прогрессивный выигрыш для каждой следующей клеточки. Например, если нам нужно заполнить 20 клеточек, то два кандидата на каждую клеточку дают только 1 048 576 возможных вариантов заполнения. А уровень ветвления, равный 3, для каждой из 20 клеточек дает в 3000 раз больше вариантов!

Выбрать возможных кандидатов (`possible_values`) для каждой клеточки можно двумя способами:

- *Локальный выбор (local count).*

  Наш алгоритм перебора с возвратом выдаст правильный результат, если процедура генерирования кандидатов на заполнение клеточки $(i, j)$ — т. е. процедура `possible_values`, действует очевидным образом и предоставляет на выбор цифры от 1 до 9, которых еще нет в той или иной строке, столбце или малом квадрате.

- *Просмотр вперед (look ahead).*

  Что будет, если для нашего текущего частичного решения существует какая-то другая пустая клеточка, для которой локальные критерии не оставляют кандидатов? В таком случае это частичное решение невозможно довести до полного. Получается, что ситуация вокруг какой-то другой клеточки делает действительное количество кандидатов для клеточки $(i, j)$ равным нулю!

  Со временем мы подойдем к этой другой клеточке, обнаружим, что для нее нет действительных кандидатов, и нам придется возвращаться назад. Но зачем вообще идти к этой клеточке, если все затраченные на это усилия окажутся напрасными? Будет намного выгоднее выполнить возврат к текущей позиции и продолжить поиск в другом направлении[1].

Для успешного отсечения непродуктивных ветвей поиска требуется просмотр вперед, позволяющий обнаружить, что выбранный путь решения является тупиковым, и возвратиться из него на новый как можно раньше.

В табл. 9.1 показано количество вызовов процедуры определения решения `is_a_solution` для всех четырех комбинаций выбора следующей клеточки и возможных кандидатов для трех разных уровней сложности головоломки судоку:

- головоломка *низкого* уровня сложности предназначена для решения человеком, а не компьютером. Моя программа решила ее без единого возврата, когда для следующей клеточки выбиралась клеточка с наименьшим количеством кандидатов;
- головоломка *средней* сложности оказалась не под силу ни одному из вышедших в финал участников Мирового чемпионата по судоку в марте 2006 года. Однако для

[1] Этот подход с просмотром вперед мог бы естественным образом следовать из подхода выбора клеточки с наименьшим количеством кандидатов, если бы было разрешено выбирать клеточки, для которых нет кандидатов. Но в моей реализации уже заполненные клеточки рассматривались как не имеющие ходов, что ограничивает выбор следующей клеточки клеточками, по крайней мере, с одним кандидатом.

моей программы потребовалось только несколько возвратов, чтобы решить эту головоломку;

- головоломка *высокого* уровня сложности, показанная на рис. 9.3, содержит только 17 заполненных клеточек. Это наименьшее известное количество заполненных клеточек на всех экземплярах задачи, которое только и дает однозначное решение.

**Таблица 9.1.** Количество шагов для получения решения при разных стратегиях отсечения

| Условие отсечения | | Уровень сложности | | |
|---|---|---|---|---|
| `next_square` | `possible_values` | Низкий | Средний | Высокий |
| произвольное значение | локальный выбор | 1 904 832 | 863 305 | программа не завершена |
| произвольное значение | просмотр вперед | 127 | 142 | 12 507 212 |
| наименьшее количество кандидатов | локальный выбор | 48 | 84 | 1 243 838 |
| наименьшее количество кандидатов | просмотр вперед | 48 | 65 | 10 374 |

Что считается головоломкой высокого уровня сложности, зависит от применяемого для ее решения эвристического алгоритма. Некоторым людям теория дается труднее, чем практическое программирование, а есть и такие, кто думает иначе. Подобным образом алгоритм *A* вполне может считать, что задача $I_1$ легче, чем задача $I_2$, в то время как алгоритм *B* видит трудность этих задач в обратном порядке.

Какие выводы мы можем сделать из этих экспериментов? Предварительное исследование дальнейших вариантов решения с целью отсечения тупиковых является самым лучшим способом разрежения пространства поиска. Без применения этой операции мы никогда не решили бы головоломку самого высокого уровня, а более легкие головоломки решили бы в тысячи раз медленнее.

Разумный подход к выбору следующей клеточки имел аналогичный эффект, хотя технически мы просто изменяли порядок выполнения работы. Но обработка клеточек с наименьшим количеством кандидатов первыми равнозначна понижению исходящей степени каждого узла дерева, а каждая заполненная клеточка уменьшает количество возможных кандидатов для последующих клеточек.

При произвольном выборе следующей клеточки решение головоломки на рис. 9.3 заняло почти час (48:44). Несомненно, моя программа решила большинство других головоломок быстрее, но головоломки судоку предназначены для решения людьми за гораздо меньшее время. Выбор клеточки с наименьшим количеством кандидатов позволил сократить время поиска более чем в 1200 раз. Теперь каждая головоломка, которую мы брали, решалась за секунды — время, требуемое, чтобы взять карандаш, если вы предпочитаете решать их вручную.

Вот так проявляется вся мощь отсечения тупиковых вариантов поиска. Использование даже простых стратегий отсечения может значительно уменьшить время исполнения.

# 9.5. История из жизни. Покрытие шахматной доски

Каждый ученый мечтает о решении классической задачи — такой, которая оставалась нерешенной в течение нескольких столетий. Есть что-то романтическое в общении с ушедшими поколениями, участии в эволюционном развитии научной мысли и оказании помощи человечеству на его пути вверх по лестнице научного прогресса.

Задача может оставаться нерешенной в течение длительного времени по разным причинам. Возможно, она такая трудная, что для ее решения требуется уникальный мощный интеллект. Или же еще не были разработаны идеи или методы, требуемые для решения такой задачи. Наконец, возможно, что задача никого не заинтересовала настолько, чтобы он всерьез занялся ею. Однажды я помог решить задачу, которая оставалась нерешенной свыше сотни лет.

Люди увлекаются игрой в шахматы в течение тысяч лет. Эффект комбинаторного взрыва был впервые зафиксирован в легенде, согласно которой изобретатель шахмат запросил у правителя в качестве награды за свое изобретение самую малость — одно зерно пшеницы на первое поле шахматной доски, два — на второе и т. д., т. е. вдвое больше на каждое следующее ($i + 1$)-е поле, чем на предыдущее $i$-е. Но когда правитель узнал, что ему придется раскошелиться на:

$$\sum_{i=1}^{64} 2^{i-1} = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615\,18$$

зерен пшеницы, то его ликование по поводу приобретения такой замечательной игры по столь низкой цене сменилось негодованием по поводу алчности и коварства изобретателя.

Отрубив изобретателю голову, правитель был первым, кто применил метод отсечения в качестве меры контроля комбинаторного взрыва.

В 1849 г. немецкий гроссмейстер Йозеф Клинг (Josef Kling) поставил вопрос, возможно ли одновременно держать под ударом все 64 поля шахматной доски восемью основными фигурами: королем, ферзем, двумя конями, двумя ладьями и двумя слонами разного цвета. При этом фигуры не держат под ударом поле, на котором они находятся. Расстановки фигур, которые держат под ударом 63 поля, подобные показанным на рис. 9.4, были известны с далеких времен, но вопрос, являются ли такие расстановки наилучшими возможными, оставался открытым.

Казалось, задача решается исчерпывающим комбинаторным перебором, но возможность решить ее таким способом зависела от размера пространства поиска.

Посмотрим, сколько существует способов расстановки на шахматной доске восьми основных шахматных фигур (короля, ферзя, двух ладей, двух слонов и двух коней). Очевидный предел таких расстановок равен

$$64!/(64 - 8)! = 178\,462\,987\,637\,760 \approx 2 \cdot 10^{14}.$$

Однако было бы неразумно надеяться на перебор более чем $10^9$ расстановок на обычном компьютере за приемлемое время.

Таким образом, чтобы решить задачу расстановок, необходимо выполнить отсечение значительного объема пространства поиска. После удаления ортогональных и диаго-

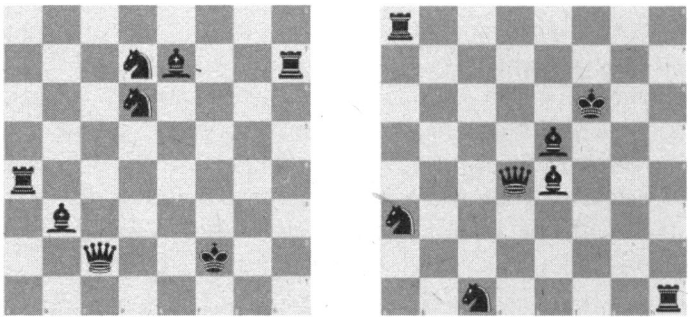


**Рис. 9.4.** Расстановки фигур, ставящие под удар 63 поля, но не все 64

нальных симметричных вариантов для ферзя останется только десять возможных позиций (рис. 9.5).

После постановки ферзя для размещения слонов остается 32×31 разных положений, затем 61×60/2 положений для ладей, 59×58/2 — для коней, а для короля — оставшиеся 57 положений. После такого отсечения остается:

$$1\ 770\ 466\ 147\ 200 \approx 1{,}8 \cdot 10^{12}$$

разных расстановок, что все равно слишком много для исчерпывающего перебора.

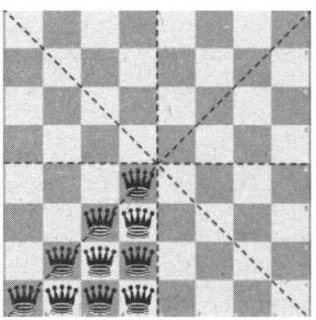


**Рис. 9.5.** Десять уникальных положений ферзя с учетом симметрии вращения и отражения

Все эти возможные расстановки можно было сгенерировать с помощью поиска с возвратом, но пространство поиска нуждалось в дополнительном значительном разрежении. Для этого нужно было найти способ, чтобы быстро выяснить, что для определенной частичной расстановки не существует возможности ее завершения, чтобы поставить под удар все 64 поля. Предположим, что мы уже разместили на доске семь фигур, которые держат под ударом все поля за исключением десяти. Далее допустим, что осталось разместить короля. Существует ли в такой ситуации поле, с которого король может держать под ударом оставшиеся девять полей? Ответ однозначно должен быть отрицательным, поскольку согласно правилам шахмат король может держать под ударом самое большее восемь полей. То есть нет смысла проверять наличие решения при расположении короля на любом из оставшихся полей.

Такая стратегия отсечения расстановок может дать большую экономию, но для ее реализации необходимо внимательно исследовать порядок постановки фигур на доску.

Каждая фигура может держать под ударом определенное максимальное количество полей: ферзь — 27, король и конь — 8, ладья — 14, а слон — 13. Хорошей стратегией может быть постановка фигур на доску в порядке убывания их влияния: *Q* (queen, ферзь), *R*1 (rook, ладья), *R*2, *B*1 (bishop, слон), *B*2, *K* (king, король), *N*1 (knight, конь), *N*2. В результате мы можем выполнять отсечение дальнейших расстановок в любое время, когда количество полей, не находящихся под боем, превышает сумму возможностей оставшихся непоставленных фигур. Эта сумма минимизируется постановкой фигур в убывающем порядке их возможностей держать поля под ударом.

Когда мы реализовали перебор с возвратом, используя эту стратегию отсечения, то отсекли свыше 95% пространства поиска. После оптимизации метода генерирования постановок фигур наша программа могла исследовать 1000 полей за секунду на современном на то время компьютере. Но и это было слишком медленно, т. к. $10^{11}/10^3 = 10^8$ секунд означало 1000 дней! Хотя мы могли бы настроить программу и ускорить время ее исполнения примерно на порядок, но в действительности нам было нужно найти способ отсечь еще больше тупиковых расстановок.

Чтобы отсечение было эффективным, требовалось устранить большое количество расстановок одним приемом, а наши предыдущие попытки в этом направлении были слишком слабыми. А если поставить на доску не восемь фигур, а больше? Очевидно, что чем больше фигур поставлено на доску, тем больше вероятность, что они будут держать под ударом все 64 поля. Но *если* большее количество фигур не держит под ударом все 64 поля, то и любое из восьмифигурных подмножеств этого множества не способно на это. Этот подход позволяет устранить огромное количество расстановок, удалив всего лишь один узел.

Так что в последней версии нашей программы узлы дерева поиска представляли расстановки, которые могли содержать любое количество фигур и больше, чем одну фигуру на одном и том же поле. Для определенной расстановки мы различали *сильную* и *слабую* атаку поля. Сильная атака соответствует обычной атаке по шахматным правилам. Поле считается слабо атакованным, если блокировка одних фигур другими игнорируется. Как можно видеть на рис. 9.6, все 64 поля можно держать под слабой атакой восемью фигурами.

Наш алгоритм выполнял два прохода. В первом проходе перечислялись расстановки, в которых каждое поле находилось под слабой атакой, а во втором список разрежался

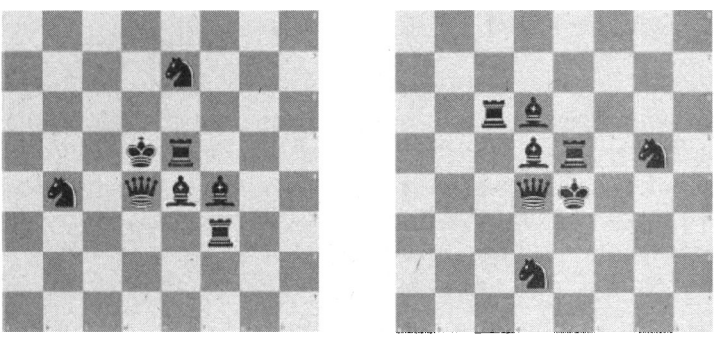


**Рис. 9.6.** Слабая атака всех 64 полей шахматной доски восемью основными фигурами

благодаря фильтрации расстановок с блокирующими фигурами. Расстановку со слабой атакой вычислить намного легче, поскольку в этом случае нет необходимости принимать во внимание блокирующие фигуры, а любое множество сильных атак является подмножеством множества слабых атак. Поэтому любую расстановку, содержащую поле под сильной атакой, можно было отсечь.

Полученная версия программы была достаточно эффективной, чтобы завершить поиск менее чем за один день. Она не нашла ни одной расстановки, удовлетворяющей первоначальным условиям задачи. Но с ее помощью мы смогли доказать, что возможно поставить под удар все поля шахматной доски, используя *семь* фигур, — при условии, что ферзь и конь могут занимать одно и то же поле. На рис. 9.7 показано соответствующее расположение фигур, причем ферзь и конь, расположенные на одном поле, обозначены белым ферзем.

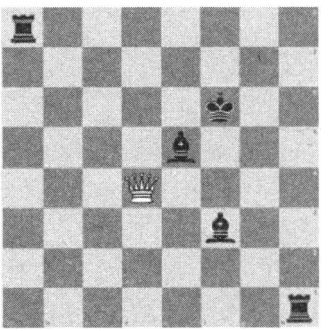


**Рис. 9.7.** Семи фигур достаточно при размещении ферзя и коня на одном поле (на этом поле сейчас расположен белый ферзь)

#### *Подведение итогов*

Использование интеллектуальной стратегии отсечения тупиковых решений может позволить с удивительной легкостью решить комбинаторные задачи, на первый взгляд кажущиеся неразрешимыми. Выполненное должным образом отсечение окажет большее воздействие на время перебора, чем любой другой фактор типа структуры данных или языка программирования.

## 9.6. Поиск методом «лучший-первый»

Исследование наилучших возможных решений перед менее обещающими — важная идея для ускорения поиска. В рассмотренной ранее (см. *разд. 9.1*) реализации поиска перебором с возвратом (процедура `backtrack`) порядок поиска определяется последовательностью элементов, создаваемых процедурой `construct_candidates`. В частности, элементы в начале массива кандидатов рассматриваются прежде элементов, расположенных дальше к концу массива. Хорошее упорядочивание кандидатов может очень сильно повлиять на время решения задачи.

До сих пор в этой главе мы рассматривали примеры, концентрирующиеся на задачах *экзистенциального* поиска, где мы ищем одно (или все) решение, удовлетворяющее

заданному множеству ограничений. В *задачах оптимизации* мы ищем решение с наименьшим или наибольшим значением некой целевой функции. Простая стратегия поиска решения для таких задач состоит в создании всех возможных решений, а затем выборе из них того, что лучше всего отвечает критерию оптимизации. Но этот подход может оказаться затратным. Намного лучше будет создавать решения последовательно, начиная с наилучшего и перемещаясь к наихудшему, и выдавать наилучшее решение, как только можно доказать, что оно действительно наилучшее.

В методе поиска «лучший-первый» (best-first), также называемом *методом ветвей и границ* (branch and bound), каждому созданному частичному решению присваивается стоимость. Эти частичные решения отслеживаются по стоимости посредством очереди с приоритетами (параметр `q` в приведенном далее листинге 9.16), что позволяет легко определить наиболее обещающее частичное решение и расширить его. Как и в случае поиска перебором с возвратом, следующее возможное частичное решение тестируется процедурой `is_a_solution`, и если оно является таковым, то вызывается процедура `process_solution`. Для определения всех способов расширения этого частичного решения вызывается процедура `construct_candidates`, и все полученные кандидаты вставляются вместе с соответствующей стоимостью в очередь с приоритетами. В листинге 9.16 приводится реализация общего поиска методом «лучший-первый» для решения задачи коммивояжера.

**Листинг 9.16. Поиск кандидатов кратчайшего пути методом «лучший-первый»**

```
void branch_and_bound (tsp_solution *s, tsp_instance *t) {
    int c[MAXCANDIDATES];      /* Кандидаты для следующей позиции */
    int nc;                    /* Счет кандидатов на следующую позицию */
    int i;                     /* Счетчик*/

    first_solution(&best_solution,t);
    best_cost = solution_cost(&best_solution, t);
    initialize_solution(s,t);
    extend_solution(s,t,1);
    pq_init(&q);
    pq_insert(&q,s);

    while (top_pq(&q).cost < best_cost) {
        *s = extract_min(&q);
        if (is_a_solution(s, s->n, t)) {
            process_solution(s, s->n, t);
        }
        else {
              construct_candidates(s, (s->n)+1, t, c, &nc);
              for (i=0; i<nc; i++) {
                  extend_solution(s,t,c[i]);
                  pq_insert(&q,s);
                  contract_solution(s,t);
              }
        }
    }
}
```

В листинге 9.17 приводится код процедур `extend_solution` и `contract_solution` для создания частичных решений, связанных с каждым новым кандидатом, и присвоения им стоимости.

**Листинг 9.17. Процедуры `extend_solution` и `contract_solution`**

```
void extend_solution(tsp_solution *s, tsp_instance *t, int v) {
    s->n++;
    s->p[s->n] = v;
    s->cost = partial_solution_lb(s,t);
}

void contract_solution(tsp_solution *s, tsp_instance *t) {
    s->n--;
    s->cost = partial_solution_lb(s,t);
}
```

Какой должна быть стоимость частичного решения? Для $n$ точек возможны $(n-1)!$ циклических перестановок, поэтому каждый маршрут можно представить в виде перестановки из $n$ элементов, начиная с 1, чтобы не было повторений. В частичных решениях создается префикс маршрута, начиная с вершины $v_1$, поэтому естественной функцией стоимости может быть сумма весов ребер по источнику этого префикса. Такая функция стоимости обладает интересным свойством: она служит *нижней границей* стоимости любого расширенного маршрута — в предположении, что все ребра имеют положительный вес.

Но будет ли первое полное решение, выданное поиском «лучший-первый», оптимальным решением? Не обязательно. Если мы взяли его из очереди с приоритетами, тогда, безусловно, более дешевого частичного решения не было. Но расширение такого частичного решения повлекло некоторые затраты — в частности, следующее ребро, добавленное к этому маршруту. Вполне возможно, что слегка более дорогостоящий частичный маршрут можно завершить, используя более легкое следующее ребро, создавая тем самым лучшее решение. Таким образом, чтобы получить оптимальное глобальное решение, необходимо исследовать снимаемые с очереди с приоритетами частичные решения до тех пор, пока они не станут более дорогостоящими, чем уже известное наилучшее решение. Обратите внимание на то, что для этого требуется, чтобы функция стоимости для частичных решений была нижней границей стоимости оптимального решения.

В противном случае далее в очереди может находиться частичное решение, которое могло бы быть расширенным до лучшего полного решения. Тогда у нас не было бы другого выбора, кроме как полностью расширить все частичные решения в очереди с приоритетами, чтобы гарантировать нахождение правильного решения.

# 9.7. Эвристический алгоритм A*

Когда функция стоимости частичного решения определяет нижнюю границу оптимального маршрута, его исследование можно прекратить, как только будет получено более дешевое решение, чем наилучшее неисследованное частичное решение. Но даже при этом поиск «лучший-первый» может занять некоторое время.

Рассмотрим частичные решения, которые мы обнаружим в процессе поиска оптимального решения задачи коммивояжера. Стоимость решения повышается с увеличением количества ребер частичного решения, поэтому частичные решения с меньшим количеством узлов всегда будут выглядеть более обещающими, чем более длинные решения, близкие к завершению. Даже самый ужасный префикс пути с $n/2$ узлами окажется, скорее всего, более дешевым, чем оптимальное решение со всеми $n$ узлами, — т. е. нам нужно развернуть все частичные решения до такой степени, пока их стоимость не превысит стоимость наилучшего полного маршрута. Эта обработка будет ужасно дорогостоящей.

*Эвристический алгоритм A** (произносится «А звезда» или «А стар» — от *англ.* A-star) представляет собой более усовершенствованный вариант поиска «лучший-первый», при котором наилучшее (самое дешевое) текущее частичное решение расширяется в каждой итерации цикла. Идея заключается в том, чтобы использовать нижнюю границу стоимости всех расширений частичного решения, которая более стойкая, чем просто стоимость текущего частичного маршрута. В результате частичные привлекательные решения будут выглядеть еще более привлекательными, чем решения с наименьшим количеством вершин.

Но как можно задать нижнюю границу стоимости полного маршрута, который содержит $n$ ребер, на основе частичного решения, содержащего $k$ вершин (и, таким образом, $(k-1)$ ребер)? Мы знаем, что в конечном итоге в решение будет добавлено $n-k+1$ дополнительных ребер. Если `minlb` обозначает нижнюю границу стоимости любого ребра — в частности, расстояние между двумя ближайшими точками, — тогда выражение $(n-k+1) \times$ `minlb` дает намного более реалистическую нижнюю границу стоимости для частичного решения. Соответствующий код приводится в листинге 9.18.

**Листинг 9.18. Вычисление стоимости решения**

```
double partial_solution_cost(tsp_solution *s, tsp_instance *t) {
    int i;                          /* Счетчик*/
    double cost = 0.0;              /* Стоимость решения */

    for (i = 1; i < (s->n); i++) {
        cost = cost + distance(s, i, i + 1, t);
    }

    return(cost);
}

double partial_solution_lb(tsp_solution *s, tsp_instance *t) {
    return(partial_solution_cost(s,t) + (t->n - s->n + 1) * minlb);
}
```

В табл. 9.2 приводится количество вычислений стоимости полного решения при нахождении оптимального решения задачи коммивояжера для нескольких вариантов поиска. Для поиска перебором с возвратом без отсечений требуется $(n - 1)!$ таких вызовов, для поиска с отсечением частичных решений требуется намного меньше вызовов и еще меньше — для поиска с отсечением по полной нижней границе. Но для алгоритмов поиска методом «лучший-первый» и его варианта A* требуется еще меньшее количество вызовов.

**Таблица 9.2.** Количество вычислений стоимости полных решений задачи коммивояжера для разных вариантов поиска. (Лучше всего в этом отношении — намного лучше, чем перебор с возвратом, — показал себя вариант A* метода «лучший-первый»)

| n | Перебор с возвратом | | | Лучший-первый | |
|---|---|---|---|---|---|
| | все | стоимость < наилучшего | нижн. гран. < наилучшего | стоимость < наилучшего | нижн. гран. < наилучшего |
| 5 | 24 | 22 | 17 | 11 | 7 |
| 6 | 120 | 86 | 62 | 28 | 20 |
| 7 | 720 | 217 | 153 | 51 | 42 |
| 8 | 5040 | 669 | 443 | 111 | 85 |
| 9 | 40 320 | 2509 | 1619 | 354 | 264 |
| 10 | 362 880 | 5042 | 3025 | 655 | 475 |
| 11 | 3 628 800 | 12 695 | 6391 | 848 | 705 |

Обратите внимание на то, что количество обнаруженных полных решений значительно уменьшает общий объем выполненной работы по поиску, которая включает даже частичные решения, отсеченные лишь за один шаг перед завершением маршрута. Но в табл. 9.2 отражается тот факт, что поиску типа «лучший-первый» может может потребоваться рассмотрение значительно меньшей части дерева поиска, чем при поиске с возвратом, даже с теми же критериями отсечения.

Поиск методом «лучший-первый» — это в какой-то степени поиск с возвратами. Недостатком поиска «лучший-первый» (обход в ширину) по сравнению с поиском перебором с возвратом (обход в глубину) является требуемый объем памяти. Объем памяти, используемый поиском перебором с возвратом (обход в глубину), пропорционален высоте дерева, но для поиска «лучший-перый» (обход в ширину) дерево должно содержать все частичные решения, более похожие на ширину дерева.

Реальной проблемой здесь является получаемый размер очереди с приоритетами для поиска «лучший-первый». Обратимся к рассмотренным ранее экспериментам с решением задачи коммивояжера. Для $n = 11$ размер очереди составляет 202 063, тогда как размер стека для поиска перебором с возвратом равен 11. Так что проблема нехватки памяти проявится быстрее, чем проблема слишком долгого времени исполнения. Чтобы получить ответ от медленной программы, нужно лишь набраться достаточно терпе-

ния, но программа, потерпевшая сбой из-за недостатка памяти, не выдаст ответа, сколько бы его ни ждать.

#### Подведение итогов

Перспективность рассмотренного частичного решения состоит не только в стоимости его самого, но также включает в себя потенциальную стоимость остатка решения. Жесткая оценка затрат решения, которая при этом также является и нижней границей стоимости, делает поиск «лучший-первый» намного более эффективным.

Эвристический алгоритм A* полезен для решения широкого круга задач — в особенности для нахождения кратчайших путей в графе от вершины $s$ к вершине $t$. Вспомним, что алгоритм Дейкстры для поиска кратчайшего пути начинает с вершины $s$ и с каждой итерацией цикла добавляет новую вершину к пути, который он считает кратчайшим. Когда граф представляет дорожную сеть на поверхности Земли, такая область известного пути должна разрастаться концентрическими кругами вокруг вершины $s$.

Но это означает, что половина этого разрастания направлена в обратную сторону от вершины $t$, просто-напросто удаляясь от цели. В поиске «лучший-первый» к расстоянию в дереве от вершины $s$ к вершине $v$ добавляется кратчайшее (по прямой линии) расстояние от каждой находящейся в дереве вершины $v$ к вершине $t$. Это дает нижнюю границу расстояния от точки $s$ к точке $t$, при этом имеется в виду рост маршрута в правильном направлении. Существование таких эвристических алгоритмов для нахождения кратчайших путей и объясняет, как онлайновые навигаторы могут так быстро выдать вам дорогу домой.

## Замечания к главе

Обсуждение поиска перебором с возвратом в значительной степени основано на моей книге «Programming Challenges» [SR03]. В частности, рассмотренная здесь процедура поиска перебором с возвратом является обобщенной версией процедуры, представленной в *главе 8* этой книги. Там также имеется мое решение знаменитой задачи о восьми ферзях, где требуется найти все расположения на шахматной доске восьми ферзей, ни один из которых не находится под ударом любого другого.

Для более подробной информации по комбинаторному поиску оптимальных позиций на шахматной доски см. статью [RHS89].

## 9.8. Упражнения

### Перестановки

1. *[3] Беспорядком* (derangement) называется такая перестановка $p$ множества $\{1, ..., n\}$, в которой ни один элемент на находится на своем правильном месте, т. е. $p_i \neq i$ для всех $1 \leq i \leq n$. Разработайте программу перебора с возвратом с применением отсечений для создания всех беспорядочных перестановок $n$ элементов.

2. *[4] Мультимножества* (multiset) могут содержать повторяющиеся элементы. При этом мультимножество из $n$ элементов может иметь меньше чем $n!$ разных перестановок. Например, мультимножество {l, l, 2, 2} имеет только шесть разных перестановок: {1, 1, 2, 2}, {1, 2, 1, 2}, {1, 2, 2, 1}, {2, 1, 1, 2}, {2, 1, 2, 1} и {2, 2, 1, 1}. Разработайте и реализуйте алгоритм для создания всех перестановок мультимножества.

3. *[5]* Для заданного положительного целого числа $n$ найдите все такие перестановки из $2n$ элементов мультиможества $S = \{1, 1, 2, 2, 3, 3, ..., n, n\}$, чтобы для каждого целого числа от 1 до $n$ количество промежуточных элементов между двумя его вхождениями равнялось значению элемента.

   Например, вот два возможных решения для ввода $n = 3$: [3, 1, 2, 1, 3, 2] и [2, 3, 1, 2, 1, 3].

4. *[5]* Разработайте и реализуйте алгоритм для проверки, являются ли два графа изоморфными по отношению друг к другу. Задача изоморфизма графов рассматривается в *разд. 19.9*. При правильном отсечении можно без проблем выполнять проверку графов, содержащих сотни вершин.

5. *[5]* Множество $\{1, 2, 3, ..., n\}$ содержит в целом $n!$ различных перестановок. Перечислив и пометив все перестановки для $n = 3$ в возрастающем лексикографическом порядке, получим такую последовательность:

   [123, 132, 213, 231, 312, 321].

   Разработайте эффективный алгоритм, который для вводов $n$ и $k$ возвращает $k$-ю из $n!$ перестановок в приведенной последовательности. В целях эффективности алгоритм не должен создавать в этом процессе первые $k$–1 перестановок.

## Перебор с возвратом

6. *[5]* Для заданного значения $n$ сгенерируйте все структурно различные деревья двоичного поиска, хранящие значения 1 ... $n$.

7. *[5]* Реализуйте алгоритм для отображения всех действительных (т. е. должным образом открытых и закрытых) последовательностей $n$ пар скобок.

8. *[5]* Сгенерируйте все возможные топологические упорядочения заданного бесконтурного ориентированного графа.

9. *[5]* Для заданного мультимножества $S$ из $n$ целых чисел и значения $t$ найдите все различные подмножества $S$, сумма элементов которых равна $t$. Например, для $S = \{4, 3, 2, 2, 1, 1\}$ и $t = 4$ сумму элементов, равную 4, будут иметь следующие четыре различные подмножества: [4], [3, 1], [2, 2] и [2, 1, 1]. Число может входить в подмножество от одного раза до количества его вхождений в исходное мультимножество $S$, а суммой может считаться одно число.

10. *[8]* Разработайте и реализуйте алгоритм для решения задачи изоморфизма подграфов. Даны графы $G$ и $H$. Существует ли такой подграф $H'$ графа $H$, для которого граф $G$ является изоморфным графу $H'$? Как ваша программа работает в следующих особых случаях изоморфизма подграфов: гамильтонов цикл, клика, независимое множество?

11. *[5] Распределение по командам* $n = 2k$ игроков заключается в разбиении игроков на две команды, каждая из которых содержит точно $k$ игроков. Например, игроков $\{A, B, C, D\}$ можно распределить по двум командам с одинаковым количеством игроков тремя разными способами: $\{\{A, B\}, \{C, D\}\}$, $\{\{A, C\}, \{B, D\}\}$ и $\{\{A, D\}, \{B, C\}\}$.

a) Создайте список 10 возможных распределений по командам $n = 6$ игроков.

b) Разработайте эффективный алгоритм перебора с возвратом для создания всех возможных распределений по командам. При этом повторяющиеся решения не допускаются.

12. *[5]* Для заданного алфавита $\Sigma$, множества запрещенных строк $S$ и длины целевой строки $n$ разработайте алгоритм для создания строки длиной $n$ из элементов алфавита $\Sigma$, не содержащей никаких подстрок из $S$. Например, для $\Sigma = \{0, 1\}$, $S = \{01, 10\}$ и $n = 4$ возможны два решения: 0000 и 1111. А вот для $S = \{0, 11\}$ и $n = 4$ решений не существует.

13. *[5]* Задача $k$ разбиений заключается в разбиении мультимножества натуральных чисел на $k$ непересекающихся подмножеств с одинаковой суммой составляющих их элементов. Разработайте и реализуйте алгоритм для решения задачи $k$ разбиений.

14. *[5]* Имеется взвешенный ориентированный граф $G$, содержащий $n$ вершин и $m$ ребер. Средний вес цикла — это сумма весов его ребер, разделенная на количество ребер. Найдите в графе $G$ цикл с минимальным средним весом.

15. *[8]* В проекте реконструкции автострады дано мультимножество $D$ из $n(n-1)/2$ расстояний. Задача заключается в том, чтобы расположить $n$ точек на линии таким образом, чтобы расстояния между парами этих точек были элементами мультимножества $D$. Например, расстояния $D = \{1, 2, 3, 4, 5, 6\}$ можно получить, расположив вторую точку на расстоянии 1 единицы от первой, третью — на расстоянии 3 единиц от второй, а четвертую — на расстоянии 2 единиц от третьей. Разработайте и реализуйте эффективный алгоритм для нахождения всех решений этой задачи. Чтобы ускорить поиск, используйте по мере возможности аддитивные ограничения. При правильном отсечении можно без проблем решать задачи, содержащие сотни узлов.

## Игры и головоломки

16. *[5] Анаграммой* называется перестановка букв слова или фразы таким образом, чтобы получилось другое слово или фраза. Иногда результаты таких перестановок поражают. Например, фраза MANY VOTED BUSH RETIRED (многие проголосовали против Буша) является анаграммой фразы TUESDAY NOVEMBER, THIRD (вторник, третье ноября)[2], что правильно отражает результат президентских выборов в США в 1992 году. Разработайте и реализуйте алгоритм создания анаграмм, используя комбинаторный поиск и словарь.

17. *[5]* Сгенерируйте все последовательности ходов конем, которые можно выполнить на шахматной доске размером $n \times n$, когда каждую клетку доски можно посетить только один раз.

18. *[5]* Доска Boggle представляет собой сетку символов размером $n \times m$. Для такой доски нужно найти все возможные слова, которые могут быть образованы последовательностью соседних символов на доске без повторений. Например, следующая доска:

```
e  t  h  t
n  d  t  i
a  i  h  n
r  h  u  b
```

[2] День президентских выборов в США в 1992 г., когда действующий президент Буш проиграл Биллу Клинтону. — *Прим. пер.*

содержит слова `tide`, `dent`, `raid` и `hide`. Разработайте алгоритм для генерирования большинства слов для доски *B* в соответствии со словарем *D*.

19. *[5]* Квадрат Бэббиджа представляет собой матрицу букв, соответствующие столбцы и строки которой содержат одинаковые слова. Например, для слова `hair` можно составить два таких квадрата:

```
h a i r | h a i r
a i d e | a l t o
i d l e | i t e m
r e e f | r o m b
```

Для заданного слова из *k* букв и словаря из *n* слов найдите все квадраты Бэббиджа, начинающиеся с этого слова.

20. *[5]* Продемонстрируйте, что можно решить любую головоломку судоку, вычислив минимальную вершинную раскраску конкретного, должным образом созданного графа, содержащего 9 × 9 + 9 вершин.

## Комбинаторная оптимизация

Для каждой из приведенных далее задач 21 по 27 реализуйте программу комбинаторного поиска для оптимального решения входного экземпляра небольшого размера. Дайте оценку практической работе вашей программы.

21. *[5]* Разработайте и реализуйте алгоритм для решения задачи уменьшения ширины ленты матрицы, рассматриваемой в *разд. 16.2*.

22. *[5]* Разработайте и реализуйте алгоритм для решения задачи максимальной приемлемости, рассматриваемой в *разд. 17.10*.

23. *[5]* Разработайте и реализуйте алгоритм для решения задачи поиска максимальной клики, рассматриваемой в *разд. 19.1*.

24. *[5]* Разработайте и реализуйте алгоритм для решения задачи минимальной вершинной раскраски, рассматриваемой в *разд. 19.7*.

25. *[5]* Разработайте и реализуйте алгоритм для решения задачи реберной раскраски, рассматриваемой в *разд. 19.8*.

26. *[5]* Разработайте и реализуйте алгоритм для решения задачи поиска разрывающего множества вершин, рассматриваемой в *разд. 19.11*.

27. *[5]* Разработайте и реализуйте алгоритм для решения задачи покрытия множества, рассматриваемой в *разд. 21.1*.

## Задачи, предлагаемые на собеседовании

28. *[4]* Напишите функцию поиска всех перестановок букв в заданной строке.

29. *[4]* Реализуйте алгоритм перечисления всех *k*-элементных подмножеств множества, содержащего *n* элементов.

30. *[5]* *Анаграммой* называется перестановка букв слова или фразы таким образом, чтобы получилось другое слово или фраза. Например, анаграммой строки `Steven Skiena` является строка `Vainest Knees`. Предложите алгоритм для создания всех анаграмм заданной строки.

31. *[5]* Каждая кнопка телефонного номеронабирателя содержит несколько букв. Напишите программу для генерирования всех возможных слов, которые можно получить из заданной последовательности нажатых кнопок (например, 145345).
32. *[7]* Имеется пустая комната и $n$ человек снаружи. На каждом этапе можно впустить одного человека в комнату или выпустить одного человека из комнаты. Можете ли вы упорядочить последовательность из $2^n$ этапов таким образом, чтобы каждая возможная комбинация людей возникла только один раз?
33. *[4]* Используя генератор случайных чисел, который генерирует случайные целые числа в диапазоне от 0 до 4 с одинаковой вероятностью, напишите генератор случайных чисел (`rng07`), который генерирует целые числа в диапазоне от 0 до 7 с одинаковой вероятностью. Укажите ожидаемое количество вызовов функции `rng04` для каждого вызова функции `rng07`.

## LeetCode

1. **https://leetcode.com/problems/subsets/**
2. **https://leetcode.com/problems/remove-invalid-parentheses/**
3. **https://leetcode.com/problems/word-search/**

## HackerRank

1. **https://www.hackerrank.com/challenges/sudoku/**
2. **https://www.hackerrank.com/challenges/crossword-puzzle/**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org**:

«Little Bishops», глава 8, задача 861.

1. «15-Puzzle Problem», глава 8, задача 10181.
2. «Tug of War», глава 8, задача 10032.
3. «Color Hash», глава 8, задача 704.

# Динамическое программирование

Наиболее трудными алгоритмическими задачами являются задачи оптимизации, в которых требуется найти решение, максимизирующее или минимизирующее целевую функцию. Классическим примером задачи оптимизации является задача коммивояжера, в которой требуется найти маршрут с минимальной стоимостью для посещения всех вершин графа. Как мы видели в *главе 1*, для решения задачи коммивояжера можно с легкостью предложить несколько алгоритмов, выдающих кажущиеся удовлетворительными решения, которые при этом *не всегда* определяют маршрут с минимальной стоимостью.

Для алгоритмов задач оптимизации требуется доказательство, что они всегда возвращают наилучшее возможное решение. «Жадные» алгоритмы, которые принимают наилучшее локальное решение на каждом шаге, обычно эффективны, но не гарантируют глобального оптимального решения. Алгоритмы поиска методом исчерпывающего перебора всегда выдают оптимальный результат, но обычно временна́я сложность таких алгоритмов чрезмерно высока.

Динамическое программирование сочетает лучшие возможности обоих подходов. Этот метод предоставляет возможность разрабатывать алгоритмы специального назначения, которые систематически исследуют все возможности (таким образом гарантируя правильность решения) и в то же время сохраняют ранее полученные промежуточные результаты (тем самым обеспечивая эффективность работы). Сохранение *последствий* всех возможных решений и систематическое использование этой информации минимизируют общий объем работы.

Если вы поймете суть динамического программирования, эта технология разработки алгоритмов, возможно, явится для вас самой удобной в плане практического применения. При этом я считаю, что алгоритмы динамического программирования часто легче разработать заново, чем искать их готовую реализацию в какой-либо книге. Однако пока вы не понимаете динамическое программирование, оно кажется вам каким-то шаманством. Поэтому, прежде чем использовать этот прием, необходимо полностью разобраться, как он работает.

Динамическое программирование — это технология для эффективной реализации рекурсивных алгоритмов посредством сохранения промежуточных результатов. Секрет ее применения заключается в определении, выдает ли простой рекурсивный алгоритм одинаковые результаты для одинаковых подзадач. Если выдает, то вместо повторения вычислений ответ каждой подзадачи можно сохранять в таблице для использования в дальнейшем, что дает возможность получить эффективный алгоритм. Начинаем мы разработку с определения и отладки рекурсивного алгоритма. И только добившись правильной работы нашего рекурсивного алгоритма, переходим к поиску мер по ускорению его работы, сохраняя результаты в матрице.

Динамическое программирование обычно является подходящим методом решения задач оптимизации в случае комбинаторных объектов, которые имеют естественный порядок организации компонентов *слева направо*. К таким объектам относятся строки символов, корневые деревья, многоугольники, а также последовательности целых чисел. Изучение динамического программирования лучше всего начинать с исследования готовых примеров. Для демонстрации практической пользы от динамического программирования в этой главе приводится несколько историй из жизни, когда оно сыграло решающую роль в решении поставленной задачи.

# 10.1. Кэширование и вычисления

По сути, динамическое программирование является компромиссом, при котором повышенный расход памяти компенсируется экономией времени. Многократное вычисление некоего значения может оказать отрицательное влияние на производительность. В таком случае для повышения производительности разумнее не повторять вычисления, а сохранять результаты первоначальных вычислений и потом обращаться к ним в случае надобности.

Экономия времени исполнения за счет повышенного расхода памяти в динамическом программировании лучше всего проявляется при рассмотрении рекуррентных соотношений, таких как числа Фибоначчи. В последующих разделах мы рассмотрим три программы для вычисления этих последовательностей.

## 10.1.1. Генерирование чисел Фибоначчи методом рекурсии

Числа Фибоначчи впервые были исследованы в XIII веке итальянским математиком Фибоначчи для моделирования размножения кроликов. Фибоначчи предположил, что количество пар кроликов, рождающихся в тот или иной месяц, равно сумме пар кроликов, рожденных в два предыдущих месяца, начиная с одной пары кроликов. Таким образом, для подсчета количества кроликов, рожденных в $n$-м месяце, он определил следующее рекуррентное соотношение:

$$F_n = F_{n-1} + F_{n-2}$$

для которого $F_0 = 0$ и $F_1 = 1$. Тогда $F_2 = 1$, $F_3 = 2$. Ряд продолжается в виде последовательности {3, 5, 8, 13, 21, 34, 55, 89, 144, ...}. Как потом выяснилось, формула Фибоначчи не очень хорошо подходит для описания размножения кроликов, но зато оказалось, что она обладает множеством интересных свойств и применений. Так как числа Фибоначчи определяются рекурсивной формулой, то для вычисления $n$-го числа Фибоначчи легко создать рекурсивную программу. Пример такого рекурсивного алгоритма на языке C приводится в листинге 10.1.

**Листинг 10.1. Рекурсивная функция для вычисления *n*-го числа Фибоначчи**

```
long fib_r(int n) {
    if (n == 0) {
        return(0);
    }
```

```
    if (n == 1) {
        return(1);
    }

    return(fib_r(n-1) + fib_r(n-2));
}
```

Ход выполнения этого рекурсивного алгоритма можно проиллюстрировать его *рекурсивным деревом*, показанным на рис. 10.1.

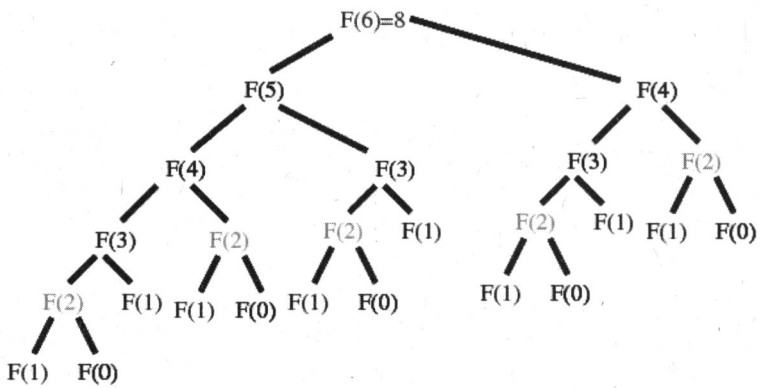



**Рис. 10.1.** Дерево для рекурсивного вычисления чисел Фибоначчи

Как и все рекурсивные алгоритмы, это дерево обрабатывается посредством обхода в глубину. Я настоятельно рекомендую проследить выполнение этого примера вручную, чтобы освежить ваши знания о рекурсии.

Обратите внимание, что число $F(4)$ вычисляется на обеих сторонах дерева, а число $F(2)$ вычисляется в этом небольшом примере целых пять раз. Полный вес этих избыточных вычислений становится ясным при исполнении программы. Для вычисления числа $F(50)$ на моем ноутбуке этой программе потребовались 4 минуты и 40 секунд. Используя предоставленный в следующем разделе алгоритм, эти числа вполне можно было бы вычислить быстрее вручную.

Сколько времени этот рекурсивный алгоритм будет вычислять число $F(n)$? Так как

$$F_{n+1} / F_n \approx \varphi = (1+\sqrt{5})/2 = 1{,}61803,$$

это означает, что $F_n > 1{,}6^n$ для достаточного большого значения $n$. Поскольку листьями нашего дерева являются только 0 и 1, то такая большая сумма означает, что у нас должно быть, по крайней мере, $1{,}6^n$ листьев или вызовов процедуры! Иными словами, эта небольшая простенькая программа имеет экспоненциальную временную сложность.

## 10.1.2. Генерирование чисел Фибоначчи посредством кэширования

Но на самом деле мы можем решить эту задачу намного эффективнее. Для этого мы явно сохраняем (или *кэшируем*) результаты вычисления каждого числа Фибоначчи $F(k)$

в таблице. Этот метод называется *запоминанием (сохранением) результатов* (memoization). Ключ к эффективной реализации рекурсивного алгоритма: прежде чем вычислять значение, мы сначала явно проверяем его наличие в таблице, таким образом избегая повторных вычислений. Соответствующий код приводится в листинге 10.2.

**Листинг 10.2. Вычисление чисел Фибоначчи с использованием кэширования**

```
#define MAXN 92         /* Наибольшее n, для которого значение F(n)
                           помещается в тип данных long */
#define UNKNOWN -1      /* Пустая ячейка */
long f[MAXN+1];         /* Массив для хранения значений fib */
long fib_c(int n) {
    if (f[n] == UNKNOWN) {
        f[n] = fib_c(n-1) + fib_c(n-2);
    }

    return(f[n]);
}

long fib_c_driver(int n) {
    int i;                          /* Счетчик */

    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = UNKNOWN;
    }

    return(fib_c(n));
}
```

Чтобы вычислить $F(n)$, мы вызываем процедуру `fib_c_driver(n)`. Первым делом эта процедура инициализирует кэш двумя известными значениями ($F(0)$ и $F(1)$) и флагом `UNKNOWN` для остальных, неизвестных, значений. Потом вызывается рекурсивный алгоритм вычисления, модифицированный для предварительной проверки на наличие вычисленного значения этого числа Фибоначчи.

Версия с кэшированием очень быстро доходит до максимального целого числа, которое можно представить типом `long integer`. Причины этого становятся понятными после изучения дерева рекурсии, показанного на рис. 10.2.

Здесь отсутствует сколь-нибудь значительное ветвление, поскольку вычисления выполняются только в левой ветви. Вызовы в правой ветви находят нужные значения в кэше и немедленно возвращают управление.

Какова временная сложность этого алгоритма? Дерево рекурсии является более информативным, чем код. Значение $F(n)$ вычисляется за линейное время (т. е. за время $O(n)$), поскольку рекурсивная функция `fib_c(k)` вызывается самое большее два раза для каждого значения $k$, где $0 \le k \le n-1$.

Этот общий метод явного кэширования (также называемый *методом табличного сохранения*, tabling) результатов вызовов рекурсивной функции для того, чтобы избежать

повторных вычислений, позволяет *максимально* использовать преимущества полного динамического программирования, что делает его заслуживающим более внимательного рассмотрения. В принципе, кэширование можно применять с любым рекурсивным алгоритмом. Но сохранение частичных результатов не принесет никакой пользы при работе таких рекурсивных алгоритмов, как быстрая сортировка, перебор с возвратами и обход в глубину, поскольку все рекурсивные вызовы в этих алгоритмах имеют разные значения параметров. Нет смысла сохранять то, что будет использовано один раз, а затем никогда больше не понадобится.

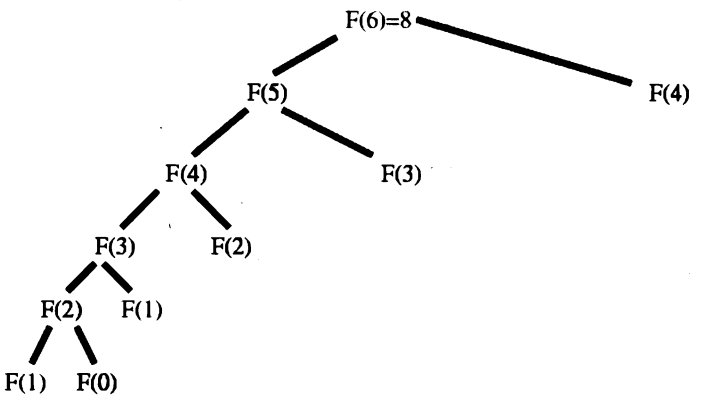



**Рис. 10.2.** Дерево вычисления чисел Фибоначчи методом кэширования

Кэширование результатов вычислений имеет смысл только в случае достаточно небольшого пространства значений параметров, когда мы можем себе позволить расходы на хранение данных. Так как аргументом рекурсивной функции `fib_c(k)` является целое число из диапазона от 0 до $n$, то кэшировать нужно только $O(n)$ значений. Нам выгодно пойти на линейные затраты памяти вместо экспоненциальных затрат времени. Но, как мы увидим далее, полностью избавившись от рекурсии, можно получить еще лучшую производительность.

#### *Подведение итогов*

Явное кэширование результатов рекурсивных процедур позволяет *максимально* использовать преимущества динамического программирования, главным достоинством которого является такое же время выполнения, что и у более элегантных решений. Если вы предпочитаете бесхитростное написание большого объема кода поиску красивого решения, вы можете не читать последующий материал.

## 10.1.3. Генерирование чисел Фибоначчи посредством динамического программирования

Число Фибоначчи $F_n$ можно вычислить за линейное время с большей эффективностью, если явно задать последовательность рекуррентных вычислений, как показано в листинге 10.3.

**Листинг 10.3. Вычисления числа Фибоначчи без рекурсии**

```
long fib_dp(int n) {
    int i;            /* Счетчик */
    long f[MAXN+1];   /* Массив для хранения значений */

    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }

    return(f[n]);
}
```

Как можно видеть, в этом варианте процедуры рекурсия совсем не используется. Мы начинаем вычисление последовательности Фибоначчи с ее наименьшего значения до заданного числа и сохраняем все промежуточные результаты. Таким образом, когда нам нужно вычислить число $F_i$, мы уже имеем требуемые для этого числа $F_{i-1}$ и $F_{i-2}$. Линейная временная сложность этого алгоритма теперь очевидна. Вычисление каждого из $n$ значений выполняется как простое суммирование двух целых чисел, общая сложность которого, как по времени, так и по памяти, равна $O(n)$.

Но более внимательное исследование процесса решения задачи показывает, что совсем не обязательно хранить все промежуточные результаты в течение вычисления всей последовательности. Так как вычисляемое значение зависит от двух аргументов, то только их и нужно сохранять. Соответствующая реализация приведена в листинге 10.4.

**Листинг 10.4. Окончательная версия процедуры вычисления чисел Фибоначчи**

```
long fib_ultimate(int n)
{
   int i;                    /* Счетчик */
   long back2=0, back1=1;    /* Последние два значения f[n] */
   long next;                /* Промежуточная сумма */
   if (n == 0) return (0);
   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
   }
   return(back1+back2);
}
```

Эта реализация понижает сложность по памяти до постоянной, при этом никак не ухудшая сложность по времени.

## 10.1.4. Биномиальные коэффициенты

В качестве другого примера устранения рекурсии указанием порядка вычислений рассмотрим вычисление *биномиальных коэффициентов*. Биномиальные коэффициенты являются наиболее важным классом натуральных чисел. В комбинаторике биномиальный коэффициент $\binom{n}{k}$ представляет количество возможных подмножеств из $k$ элементов множества из $n$ элементов.

Каким образом находятся биномиальные коэффициенты? Поскольку

$$\binom{n}{k} = \frac{n!}{k(n-k)!},$$

то их значения можно получить, вычисляя факториалы. Но этот метод имеет серьезный недостаток. Промежуточные вычисления могут вызвать арифметическое переполнение, даже если конечный результат не превышает максимальное допустимое целое число в компьютере.

Более надежным способом вычисления биномиальных коэффициентов является использование неявного рекуррентного соотношения, используемого для построения треугольника Паскаля:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Здесь каждый элемент строки является суммой двух элементов слева и справа от него в предшествующей строке. Это подразумевает следующее рекуррентное соотношение:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Почему эта формула дает правильный результат? Рассмотрим, входит ли $n$-й элемент в одно из подмножеств $\binom{n}{k}$ из $k$ элементов. Если входит, то мы можем завершить создание подмножества, выбрав другие $k-1$ элементов из оставшихся $n-1$ элементов множества. Если же не входит, то нам нужно выбрать все $k$ элементов из оставшихся $n-1$ элементов множества. Эти два случая не пересекаются и включают в себя все возможности, поэтому в сумме учитываются все подмножества из $k$ элементов.

Для любого рекуррентного соотношения требуется база. Какие значения биномиальных коэффициентов нам известны без вычислений? Левая часть формулы легко приводится к $\binom{m}{0}$. Сколько можно создать подмножеств, содержащих 0 элементов, из некого множества? Ровно одно — пустое. Если это не кажется вам убедительным, с равным успехом в качестве базы можно принять $\binom{m}{1} = m$. Правая часть формулы легко приво-

дится к $\binom{m}{m}$. Сколько можно создать подмножеств, содержащих $m$ элементов, из множества, содержащего $m$ элементов? Ровно одно — первоначальное полное множество. Эти базовые экземпляры и рекуррентное соотношение определяют все интересующие нас биномиальные коэффициенты.

| $n$ / $k$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | A | | | | | |
| 1 | B | G | | | | |
| 2 | C | 1 | H | | | |
| 3 | D | 2 | 3 | I | | |
| 4 | E | 4 | 5 | 6 | J | |
| 5 | F | 7 | 8 | 9 | 10 | K |

*а*

| $n$ / $k$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | | | | | |
| 1 | 1 | 1 | | | | |
| 2 | 1 | 2 | 1 | | | |
| 3 | 1 | 3 | 3 | 1 | | |
| 4 | 1 | 4 | 6 | 4 | 1 | |
| 5 | 1 | 5 | 10 | 10 | 5 | 1 |

*б*

**Рис. 10.3.** *а* — порядок вычисления биномиальных коэффициентов в $M$ [5, 4]. Условия инициализации помечены буквами A–K, а рекуррентные вычисления — числами 1–10; *б* — содержимое матрицы после оценки

Порядок вычисления этого рекуррентного соотношения показан на рис. 10.3. Инициализированные ячейки таблицы помечены буквами от A до K, обозначающими порядок, в котором им были присвоены значения. Каждой неинициализированной ячейке присваивается значение, равное сумме значений двух ячеек из предыдущего ряда: той, что непосредственно над ней, и той, что сверху и слева. Цифры от 1 до 10, маркирующие треугольник ячеек, обозначают порядок вычисления подмножества $\binom{5}{4} = 5$ с помощью кода, представленного в листинге 10.5.

**Листинг 10.5. Вычисление биномиального коэффициента**

```
long binomial_coefficient(int n, int k) {
    int i, j;                   /* Счетчики  */
    long bc[MAXN+1][MAXN+1]; /* Таблица биномиальных коэффициентов */

    for (i = 0; i <= n; i++) {
        bc[i][0] = 1;
    }

    for (j = 0; j <= n; j++) {
        bc[j][j] = 1;
    }

    for (i = 2; i <= n; i++) {
        for (j = 1; j < i; j++) {
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
        }
    }
```

```
    return(bc[n][k]);
}
```

Внимательно изучите эту функцию, чтобы понять, как она работает. Далее в этой главе мы будем уделять больше внимания вопросам формулирования и анализа соответствующей рекуррентности, чем технике работы с таблицами.

## 10.2. Поиск приблизительно совпадающих строк

Поиск совпадающих комбинаций символов в текстовых строках является задачей, важность которой не подлежит сомнению. В *разд. 6.7* были представлены алгоритмы для поиска точных совпадений комбинаций символов — т. е. выяснения, где именно в текстовой строке $T$ находится искомая подстрока $P$. Но, к сожалению, жизнь не всегда так проста. Например, слова, как в тексте, так и в искомой строке, могут быть написаны с ошибками, вследствие чего точного совпадения не получится. Эволюционные изменения в геномных последовательностях или языковых конструкциях приводят к тому, что мы вводим в строке поиска `Не убей`, когда нужно найти `Не убий`.

Каким образом можно нейтрализовать орфографические ошибки, чтобы найти подстроку, наиболее близкую к искомой? Для выполнения поиска неточно совпадающих строк нам нужно сначала определить функцию стоимости, с помощью которой мы будем выяснять, насколько далеко две строки находятся друг от друга, т. е. измерять расстояние между парами строк. Для этого надо учитывать количество *изменений*, которые придется выполнить, чтобы преобразовать одну строку в другую. Возможны такие действия:

- *замена* — заменяет один символ строки $P$ другим символом. Например, `shot` превращается в `spot`;
- *вставка* — вставляет один символ в строку $P$ так, чтобы она совпадала с подстрокой в тексте $T$. Например, `ago` превращается в `agog`;
- *удаление* — удаляет один символ из строки $P$ так, чтобы она совпадала с подстрокой в тексте $T$. Например, `hour` превращается в `our`.

Для правильной постановки вопроса сходства строк нам нужно установить стоимость каждой из таких операций преобразования. Присваивая каждой операции стоимость, равную 1, мы определяем *расстояние редактирования* между двумя строками. Задача нечеткого сравнения строк возникает во многих приложениях и подробно рассматривается в *разд. 21.4*.

Нечеткое сравнение строк может казаться трудной задачей, поскольку требуется решить, где именно в строке-образце и в тексте будет лучше всего выполнить сложную последовательность операций вставки и/или удаления. Для ее решения мы попробуем подойти к рассмотрению задачи с другого конца. Какая информация нам понадобится для правильного выбора последней операции? Что может случиться при сравнении с последним символом каждой строки?

## 10.2.1. Применение рекурсии для вычисления расстояния редактирования

При создании рекурсивного алгоритма можно использовать то обстоятельство, что либо последний символ в строке совпадет с искомым, либо нам придется его заменить, удалить или добавить. Как можно видеть на рис. 10.4, другого возможного выбора не существует.




**Рис. 10.4.** В одной операции редактирования строки над последним символом можно выполнить одно из следующих действий: замену, вставку или удаление

В результате отсечения символов, участвующих в операции редактирования последнего символа, останется пара более коротких строк. Пусть $i$ и $j$ будут индексами последних символов префиксов строк $P$ и $T$ соответственно. В результате последней операции образуются три пары более коротких строк, соответствующих совпадающим строкам либо строкам, полученным после замены, добавления или удаления. *Если* бы нам была известна стоимость редактирования этих более коротких строк, мы могли бы решить, какая опция дает наилучшее решение, и соответственно выбрать эту опцию. Оказывается, с помощью рекурсии мы *можем* узнать эту стоимость.

Пусть $D[i, j]$ — минимальное количество различий между подстроками $P_1P_2 \ldots P_i$ и $T_1T_2 \ldots T_j$. Иными словами, $D[i, j]$ представляет самый дешевый из трех возможных способов расширения более коротких строк:

- если ($P_i = T_j$), тогда $D[i-1, j-1]$, в противном случае $D[i-1, j-1] + 1$. Это означает, что, в зависимости от того, одинаковы ли последние символы строк, мы получаем совпадение $i$-го и $j$-го символов или заменяем их. В более общем смысле стоимость замены одного символа можно получить при помощи функции `match(Pi,Tj)`;
- $D[i, j-1] + 1$. Это означает, что в тексте имеется дополнительный символ, который нужно учесть, поэтому указатель позиции в строке образца не продвигается и выполняется вставка символа. В более общем смысле стоимость вставки одного символа можно получить при помощи функции `indel(Tj)`;
- $D[i-1, j] + 1$. Это означает, что в строке образца имеется лишний символ, который нужно удалить, поэтому указатель позиции в тексте не продвигается и выполняется удаление символа. В более общем смысле стоимость удаления одного символа можно получить при помощи функции `indel(Pi)`.

В листинге 10.6 представлена программа вычисления стоимости редактирования.

**Листинг 10.6. Вычисление стоимости редактирования методом рекурсии**

```
#define MATCH 0     /* Символ перечислимого типа для совпадения */
#define INSERT 1    /* Символ перечислимого типа для вставки */
#define DELETE 2    /* Символ перечислимого типа для удаления */
```

```
int string_compare_r(char *s, char *t, int i, int j) {
    int k;                  /* Счетчик */
    int opt[3];             /* Стоимость трех опций */
    int lowest_cost;        /* Самая низкая стоимость */

    if (i == 0) {           /* indel - стоимость вставки или удаления*/
        return(j * indel(' '));
    }

    if (j == 0) {
        return(i * indel(' '));
    }
                            /* match - стоимость совпадения/замены */

    opt[MATCH] = string_compare_r(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare_r(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare_r(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
    for (k = INSERT; k <= DELETE; k++) {
        if (opt[k] < lowest_cost) {
            lowest_cost = opt[k];
        }
    }

    return(lowest_cost);
}
```

Легко убедиться, что эта программа абсолютно корректна. Однако работает она недопустимо медленно. На моем компьютере сравнение двух строк длиной в 11 символов занимает несколько секунд, а чуть более длинные строки обрабатываются целую вечность.

Почему этот алгоритм работает так медленно? Потому что он вычисляет одни и те же значения по нескольку раз, в результате чего имеет экспоненциальную временную сложность. В каждой позиции строки происходит тройное ветвление рекурсии, вследствие чего количество вызовов растет со скоростью $3^n$, а на самом деле даже быстрее, поскольку при большинстве вызовов сокращается только один из двух индексов, а не оба.

## 10.2.2. Применение динамического программирования для вычисления расстояния редактирования

Так как же сделать этот алгоритм пригодным для использования на практике? В решении этого вопроса важным является то обстоятельство, что при большинстве рекурсивных вызовов выполняются вычисления, которые уже были выполнены. Откуда нам это известно? Возможно только $|P| \cdot |T|$ однозначных рекурсивных вызовов, поскольку именно столько имеется разных пар $(i, j)$, передаваемых в качестве параметров. Сохранив вычисленные значения для каждой из этих $(i, j)$ пар в таблице, мы можем при необходимости извлечь их из этой таблицы, вместо того, чтобы вычислять их снова.

Таблица для хранения вычисленных значений представляет собой двумерную матрицу $m$, каждая из $|P| \cdot |T|$ ячеек которой содержит значение стоимости оптимального решения подзадачи, а также родительское поле, объясняющее, как мы попали в эту ячейку. Объявление структуры таблицы приведено в листинге 10.7.

**Листинг 10.7. Структура таблицы для вычисления стоимости редактирования**

```
typedef struct {
   int cost;           /* Стоимость попадания в эту ячейку */
   int parent;         /* Родительская ячейка */
} cell;
cell m[MAXLEN+1][MAXLEN+1]; /* Таблица динамич. программирования */
```

Между динамической и рекурсивной версиями реализации алгоритма поиска неточно совпадающих строк имеются три различия. Во-первых, промежуточные значения получаются из таблицы, а не в результате рекурсивных вызовов. Во-вторых, обновляется содержимое поля родительского указателя каждой ячейки, что в дальнейшем позволит восстановить последовательность редактирования. В-третьих, в реализации используется более общая функция `goal_cell()`, а не просто возвращается значение `m[|P|][|T|].cost`. Как следствие, мы можем применять эту процедуру для решения более обширного класса задач. Реализация динамической версии алгоритма поиска неточно совпадающих строк представлена в листинге 10.8.

**Листинг 10.8. Вычисление стоимости редактирования**

```
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;            /* Счетчики */
    int opt[3];             /* Стоимость трех вариантов */

    for (i = 0; i <= MAXLEN; i++) {
       row_init(i, m);
        column_init(i, m);
    }

    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
    }
```

```
    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}
```

Здесь строки и индексы используются несколько необычным образом. В частности, полагается, что в начало каждой строки был добавлен пробел, вследствие чего первый значащий символ строки `s` находится в позиции `s[1]`. Это позволяет нам синхронизировать индексы матрицы с индексами строк. Вспомните, что нам нужно выделить нулевую строку и столбец матрицы `m` для хранения граничных значений, совпадающих с пустым префиксом. В качестве альтернативы мы бы могли оставить входные строки без изменений и откорректировать должным образом индексы.

Для определения значения ячейки $(i, j)$ требуются три готовых значения из матрицы `m`, в частности, из ячеек $m(i-1, j-1)$, $m(i, j-1)$ и $m(i-1, j)$. Подойдет любой порядок вычислений, удовлетворяющий этому условию, в том числе построчный, используемый в этой программе[1]. Два вложенных цикла в действительности вычисляют значение `m` для каждой пары префиксов строки, по одной строке за раз. Вспомним, что в позиции `s[1]` и `t[1]` этих строк добавлен первый символ каждой входной строки, так что длина (`strlen`) этих строк на один символ больше длин соответствующих входных строк.

В качестве примера в таблице, показанной на рис. ЦВ-10.5, приводится матрица стоимости преобразования строки $P$ = `thou shalt` в строку $T$ = `you should` за пять шагов. Я настоятельно рекомендую вам вычислить матрицу этого примера вручную, чтобы получить четкое представление, как именно работает динамическое программирование.

## 10.2.3. Восстановление пути

Функция сравнения строк возвращает стоимость оптимального выравнивания, но не выполняет собственно выравнивание. Нам полезно знать, что для преобразования строки `thou shalt` в строку `you should` требуется только пять операций редактирования, но было бы еще полезнее знать последовательность этих операций.

Возможные решения определенной задачи динамического программирования описываются путями в матрице динамического программирования, начинающимися с первоначальной конфигурации (пары пустых строк (0, 0)) и заканчивающимися конечным требуемым состоянием (парой заполненных строк ($|P|$,$|T|$)). Ключом к созданию решения такой задачи является реконструкция решений, принимаемых на каждом шаге оптимального пути, ведущего к целевому состоянию. Эти решения записаны в поле указателя на родителя `parent` каждой ячейки массива.

Нужные решения можно воспроизвести, выполнив проход по решениям в обратном направлении от целевого состояния и следуя указателям на родительские ячейки массива, пока не придем к начальной ячейке пути решения задачи, аналогично реконст-

[1] Допустим, мы создадим граф с вершиной для каждой ячейки матрицы и ориентированным ребром $(x, y)$, означающим, что для вычисления значения ячейки $y$ нужно значение ячейки $x$. Любая топологическая сортировка на получившемся бесконтурном ориентированном графе (кстати, почему это будет бесконтурный ориентированный граф?) определяет приемлемый порядок вычислений.

рукции пути, вычисленному обходом в ширину или алгоритмом Дейкстры. Поле указателя на родителя `parent` ячейки $m[i, j]$ содержит информацию о типе операции, выполненной в этой ячейке: `MATCH`, `INSERT` или `DELETE`. Обратная трассировка пути решения в таблице преобразования строки `thou_shalt` в строку `you_should` (рис. ЦВ-10.6) выдает нам последовательность операций редактирования `DSMMMMMISMS`. Это означает, что мы удаляем первую букву `t`, заменяем букву `h` буквой `y`, оставляем без изменений следующие пять букв, после чего вставляем букву `o`, заменяем букву `a` буквой `u` и, наконец, заменяем букву `t` буквой `d`.

Трассировка указателей на родительские ячейки восстанавливает решение в обратном порядке. Но с помощью рекурсии мы можем восстановить прямой порядок решения. Соответствующий код приведен в листинге 10.9.

**Листинг 10.9. Восстановление решения в прямом порядке**

```
void reconstruct_path(char *s, char *t, int i, int j,
                      cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1, m);
        insert_out(t, j);
        return;
    }

    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i-1, j, m);
        delete_out(s, i);
        return;
    }
}
```

Для многих задач, включая задачу вычисления расстояния редактирования, решение можно восстановить из матрицы стоимости, не прибегая к явному сохранению предыдущих операций в массиве. Для задачи вычисления расстояния редактирования нам нужно выполнить трассировку назад, начиная со значений стоимости трех возможных родительских ячеек и соответствующих символов строки, чтобы восстановить операцию, в результате выполнения которой мы оказались в текущей ячейке при той же стоимости. Но более аккуратным и легким подходом будет явно сохранять операции.

## 10.2.4. Разновидности расстояния редактирования

Процедуры сравнения строк (`string_compare`) и восстановления пути (`reconstruct_path`) вызывают несколько функций, которые еще не были определены. Эти функции можно разбить на четыре категории.

- *Инициализация таблицы.*

  Функции `row_init()` и `column_init()` инициализируют нулевые строку и столбец таблицы соответственно. В задаче вычисления расстояния редактирования ячейки ($i$, 0) и (0, $i$) соответствуют сравнению строк длиной $i$ с пустой строкой. Для этого требуется ровно $i$ вставок/удалений, поэтому код этих функций очевиден (листинг 10.10).

**Листинг 10.10. Процедуры инициализации строк и столбцов таблицы**

```
row_init(int i)
{
   m[0][i].cost = i;
   if (i>0)
      m[0][i].parent = INSERT;
   else
      m[0][i].parent = -1;
}
column_init(int i)
{
   m[i][0].cost = i;
   if (i>0)
      m[i][0].parent = DELETE;
   else
      m[i][0].parent = -1;
}
```

- *Стоимость операций.*

  Функции `match(c,d)` и `indel(c)` возвращают стоимость преобразования символа `c` в символ `d` и вставки/удаления символа `c` соответственно. Для стандартной задачи вычисления расстояния редактирования функция `match()` возвращает 0, если символы одинаковые, и 1 в противном случае. Функция `indel()` всегда возвращает 1, независимо от аргумента. Но можно также использовать функции стоимости, специфичные для конкретных приложений. Такие функции могут быть менее взыскательными к заменяемым символам, расположенным друг возле друга на стандартной раскладке клавиатуры, или символам, которые выглядят или звучат похоже.

  Общая реализация функций стоимости показана в листинге 10.11.

**Листинг 10.11. Функции стоимости**

```
int match(char c, char d)
{
   if (c == d) return(0);
```

```
    else return(1);
}
int indel(char c)
{
    return(1);
}
```

- *Определение целевой ячейки.*

Функция `goal_cell()` возвращает индексы ячейки, обозначающей конечную точку решения. В случае задачи вычисления расстояния редактирования это значение всегда определяется длиной двух входных строк. Но другие приложения, которые мы вскоре рассмотрим, не имеют фиксированного расположения решений.

Общая реализация функции определения местонахождения целевой ячейки приводится в листинге 10.12.

**Листинг 10.12. Функция определения местонахождения целевой ячейки**

```
void goal_cell(char *s, char *t, int *i, int *j)
{
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
}
```

- *Операция обратной трассировки.*

Функции `match_out()`, `insert_out()` и `delete_out()` выполняют соответствующие действия для каждой операции редактирования при трассировке решения. В случае задачи вычисления расстояния редактирования это означает вывод названия операции или обрабатываемого символа, в зависимости от требований приложения.

Общая реализация функций трассировки решения приводится в листинге 10.13.

**Листинг 10.13. Функции трассировки решения**

```
insert_out(char *t, int j)
{
    printf("I");
}
delete_out(char *s, int i)
{
    printf("D");
}

match_out(char *s, char *t,
int i, int j)
{
    if (s[i]==t[j]) printf("M");
    else printf("S");
}
```

Для задачи вычисления расстояния редактирования эти функции достаточно простые. Но следует признать, что правильное выполнение операций получения граничных состояний и манипулирования индексами является трудной задачей. Хотя при должном понимании применяемого метода алгоритмы динамического программирования легко разрабатывать, для правильной реализации деталей требуется внимательно продумать и всесторонне протестировать предлагаемое решение.

На первый взгляд может показаться, что для такого простого алгоритма потребовалось слишком много вспомогательной работы. Но только слегка модифицировав эти общие функции, мы можем решить несколько важных задач — таких как особые случаи задачи вычисления расстояния редактирования.

- *Поиск подстроки в тексте.*

  Допустим, что мы хотим найти в тексте $T$ подстроку, приблизительно совпадающую с подстрокой $P$. Например, будем искать строку `Skiena` и возможные ее варианты (`Skienna`, `Skena`, `Skeena`, `Skina` и т. п.). Поиск с помощью нашей первоначальной функции вычисления расстояния редактирования будет малочувствительным, поскольку большая часть стоимости любого редактирования будет определяться удалением всех фрагментов текста, не совпадающих со строкой `Skiena`. В нашем случае оптимальным решением будет поиск всех разбросанных в тексте совпадений с `...S...k...i...e...n...a...` и удаление всего остального.

  Нам требуется такой способ вычисления расстояния редактирования, в котором стоимость начала совпадения не зависит от местонахождения в тексте, чтобы учитывались и совпадения в середине текста. Теперь целевое состояние находится не обязательно в конце строк, а в таком месте текста, где совпадение со всем образцом имеет минимальную стоимость. Модифицировав эти две функции, мы получим требуемое правильное решение (листинг 10.14).

**Листинг 10.14. Модифицированные функции для поиска неточно совпадающих строк**

```
void row_init(int i, cell m[MAXLEN+1][MAXLEN+1]) {
    m[0][i].cost = 0;          /* ИЗМЕНЕНИЕ */
    m[0][i].parent = -1;       /* ИЗМЕНЕНИЕ */
}

void goal_cell(char *s, char *t, int *i, int *j) {
    int k; /* counter */
    *i = strlen(s) - 1;
    *j = 0;

    for (k = 1; k < strlen(t); k++) {
        if (m[*i][k].cost < m[*i][*j].cost) {
            *j = k;
        }
    }
}
```

- *Самая длинная общая подпоследовательность.*

  Допустим, что нам нужно найти самую длинную последовательность разбросанных символов, общую для обеих строк, с учетом их исходного упорядочения относительно друг друга. (Эта задача рассматривается в *разд. 21.8*). Например, если ли что-либо общее между демократами и республиканцами? Безусловно, есть! Строки `democrats` и `republicans` имеют *общую самую длинную подпоследовательность*: `ecas`.

  Общая подпоследовательность представляет собой последовательность всех разбросанных совпадающих символов в обеих строках. Чтобы максимизировать количество таких совпадений, требуется предотвратить замену несовпадающих символов. Когда замена символов запрещена, избавиться от несовпадающих символов можно только с помощью операций вставки и удаления. Для получения самого дешевого выравнивания будет выполнено наименьшее количество таких операций, поэтому в ней должна сохраниться самая длинная общая подпоследовательность. Чтобы получить требуемое выравнивание, мы модифицируем функцию стоимости совпадений, чтобы повысить стоимость замены символов (листинг 10.15).

**Листинг 10.15. Модифицированная функция стоимости совпадений**

```
int match(char c, char d) {
    if (c == d) {
        return(0);
    }
    return(MAXLEN);
}
```

  В действительности, чтобы сделать замену полностью непривлекательной операцией редактирования, будет достаточным сделать ее стоимость выше, чем совместная стоимость вставки и удаления.

- *Максимальная монотонная подпоследовательность.*

  Числовая последовательность является *монотонно возрастающей*, если каждый следующий элемент этой последовательности превышает предыдущий. Задача *максимальной монотонной подпоследовательности* состоит в удалении из входной строки $S$ наименьшего количества элементов с целью получения монотонно возрастающей подпоследовательности. Например, для последовательности `243517698` максимальной монотонной подпоследовательностью является `23568`.

  По сути, это просто задача поиска самой длинной общей подпоследовательности, где второй строкой являются элементы строки $S$, отсортированные в возрастающем порядке: `123456789`. Любая из этих двух общих последовательностей должна, во-первых, содержать символы в соответствующем порядке в строке $S$ и, во-вторых, содержать только символы с возрастающими номерами позиций в последовательности упорядочивания, так что более длинная из этих последовательностей и является решением. Конечно же, этот подход можно модифицировать для поиска самой длинной убывающей последовательности, просто изменив порядок сортировки на обратный.

Как можно видеть, базовую процедуру вычисления расстояния редактирования можно с легкостью приспособить для решения многих интересных задач. Секрет заключается в понимании, что конкретная задача является всего лишь частным случаем общей задачи нечеткого сравнения строк.

Внимательный читатель может заметить, что для вычисления затрат на выравнивание не требуется содержание всех $O(mn)$ ячеек. Если мы будем вычислять рекуррентное соотношение, заполняя столбцы матрицы слева направо, то нам никогда не потребуется более двух столбцов для хранения всей информации, требуемой для завершения этого вычисления. Таким образом, для вычисления рекуррентного соотношения достаточно объема памяти $O(m)$, при этом сложность по времени остается прежней. Это хорошо, но, к сожалению, не имея полной матрицы, мы не можем восстановить выравнивание.

Экономия памяти является важным моментом динамического программирования. Так как объем памяти на любом компьютере ограничен, то сложность по памяти, равная $O(nm)$, является более узким местом, чем такая же сложность по времени. К счастью, эта проблема решается с помощью алгоритма «разделяй и властвуй», который выполняет выравнивание за время $O(nm)$, но при этом требует память объемом только $O(m)$. Этот алгоритм рассматривается в *разд. 21.4*.

## 10.3. Самая длинная возрастающая подпоследовательность

Решение задачи посредством динамического программирования состоит из трех шагов:

1. Сформулировать требуемое решение в виде рекуррентного соотношения или рекурсивного алгоритма.
2. Показать, что количество разных значений параметра, принимаемых рекуррентностью, ограничено полиномиальной функцией (будем надеяться, небольшой степени).
3. Указать порядок вычисления рекуррентного соотношения с тем, чтобы частичные результаты были всегда доступными, когда они требуются.

Чтобы разобраться в этих деталях, рассмотрим разработку алгоритма поиска самой длинной монотонно возрастающей подпоследовательности в последовательности из $n$ чисел. Вы наверняка заметите, что эта задача была описана как частный случай задачи вычисления расстояния редактирования в *разд. 10.2.4*, где она называется задачей поиска *максимальной монотонной подпоследовательности*. Тем не менее будет полезно рассмотреть разработку ее решения с самого начала. В действительности, алгоритмы динамического программирования часто легче разработать сначала, чем искать существующее решение.

Возрастающая последовательность отличается от *серии*, в которой элементы физически находятся рядом друг с другом. Выбранные элементы обеих структур должны быть отсортированы в возрастающем порядке слева направо. Рассмотрим, например, последовательность $S = (2, 4, 3, 5, 1, 7, 6, 9, 8)$.

Самая длинная возрастающая подпоследовательность последовательности $S$ состоит, как можно видеть, из пяти символов — например: (2, 3, 5, 6, 8). На самом деле подпоследовательностей этой длины в последовательности $S$ имеется восемь (попробуйте найти их). Есть и четыре серии увеличивающейся длины: (2, 4), (3, 5), (1, 7) и (6, 9).

Поиск самой длинной возрастающей *серии* в числовой последовательности является простой задачей. Так что разработка алгоритма с линейным временем исполнения не должна вызвать особых трудностей. Но задача поиска самой длинной возрастающей подпоследовательности значительно сложнее. Как мы можем определить, какие разбросанные элементы пропустить?

Чтобы применить динамическое программирование, нам нужно создать рекуррентное соотношение, которое вычисляет длину самой длинной последовательности. Чтобы найти подходящее рекуррентное соотношение, задайте себе вопрос, какая информация о первых $n - 1$ элементах последовательности $S = (s_1, ..., s_n)$ помогла бы найти решение для всей последовательности?

- Длина $L$ самой длинной возрастающей подпоследовательности последовательности $(s_1, s_2, ..., s_{n-1})$ кажется полезной информацией. Более того, это будет длина самой длинной возрастающей последовательности в $S$, если только $s_n$ не предоставит возрастающую последовательность такой же длины.

  К сожалению, эта длина $L$ не является достаточной информацией для получения полного решения. Допустим, что каким-то образом мы узнали, что самая длинная возрастающая подпоследовательность последовательности $s_1, s_2, ..., s_{n-1}$ содержит пять символов и что $s_n = 8$. Будет ли длина самой длинной возрастающей подпоследовательности последовательности $S$ равняться 5 или 6? Это зависит от того, заканчивается ли последовательность длиной в 5 символов значением, меньшим чем 8.

- Нам необходимо знать количество символов в самой длинной последовательности, которую расширит $s_n$. Чтобы быть уверенным в том, что мы знаем это, нам в действительности нужно знать количество символов в самой длинной последовательности, оканчивающейся *любым* возможным значением для $s_n$.

Эти положения предоставляют нам базовую идею для создания рекуррентного соотношения. Определяем $L_i$ как количество символов самой длинной последовательности, заканчивающейся на $s_i$. Самая длинная возрастающая подпоследовательность, содержащая число $s_n$, получается в результате добавления этого числа в конец самой длинной возрастающей последовательности слева от $n$ и оканчивающейся числом, меньшим чем $s_n$. Длина $L_i$ вычисляется с помощью следующего рекуррентного соотношения:

$$L_i = 1 + \max_{\substack{0 \le j < i \\ s_j < s_i}} L_j,$$

$$L_0 = 0.$$

Приведенные значения определяют количество символов в самой длинной возрастающей последовательности, заканчивающейся каждым элементом последовательности. Количество символов в самой длинной возрастающей подпоследовательности полной последовательности $S$ можно выразить формулой

$$L = \max_{1 \le i \le n} L_i,$$

поскольку самая лучшая последовательность должна когда-нибудь закончиться. В табл. 10.1 приводятся данные для рассматриваемого примера.

**Таблица 10.1.** Данные для задачи поиска самой длинной возрастающей подпоследовательности

| Индекс $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Последовательность $s_i$ | 2 | 4 | 3 | 5 | 1 | 7 | 6 | 9 | 8 |
| Длина $L_i$ | 1 | 2 | 2 | 3 | 1 | 4 | 4 | 5 | 5 |
| Предшественник $p_i$ | – | 1 | 1 | 2 | – | 4 | 4 | 6 | 6 |

Какую вспомогательную информацию нам следует сохранить, чтобы восстановить саму последовательность, а не только ее длину? Для каждого элемента $s_i$ сохраняется его предшественник, а именно индекс $p_i$ элемента, непосредственно предшествующего $s_i$ в самой длинной возрастающей последовательности, заканчивающейся на $s_i$. Так как все эти указатели направлены влево, то самую длинную последовательность можно с легкостью восстановить, начав с ее последнего значения и следуя указателям обратно, чтобы воссоздать другие элементы последовательности.

Какова временная сложность этого алгоритма? Если каждое из $n$ значений $L_i$ вычисляется путем сравнения $s_i$ с $n$ значениями слева от него (где $n \geq i - 1$), то общее время этого анализа будет равно $O(n^2)$. На самом деле, умело используя словарные структуры данных, это рекуррентное соотношение можно вычислить за время $O(n \lg n)$. Поскольку простое рекуррентное соотношение легче поддается программированию, лучше начать с его реализации.

#### Подведение итогов

Когда у вас будет достаточно опыта динамического программирования, то такие алгоритмы вам станет легче создавать с нуля, чем искать готовое подходящее решение.

## 10.4. История из жизни. Сжатие текста для штрихкодов

Инджиун (Ynjiun) провел лазерной указкой по рваным и смятым кускам этикетки со штрихкодом. Через несколько секунд система выдала ответ, сопровождаемый приятным звуковым сигналом. Он победно усмехнулся: «Практически безотказно».

Мне показывали свои достижения работники научно-исследовательского центра компании Symbol Technologies, ведущего мирового производителя оборудования для сканирования штрихкодов. Хотя мы принимаем штрихкоды как должное, работа с ними требует на удивление сложной технологии. Надобность в штрихкодах существует потому, что обычные системы оптического распознавания символов не обеспечивают достаточной надежности для операций с товарно-материальными ценностями. Технология штрихкодов, знакомая каждому из нас по их присутствию на каждой пачке овсянки и упаковке жевательной резинки, позволяет закодировать десятизначный но-

мер с уровнем коррекции, делающим практически невозможными ошибки при сканировании, даже если консервная банка со штрихкодом перевернута вверх ногами или деформирована. Иногда кассиру не удается отсканировать штрихкод, но если вы слышите характерный сигнал считывателя, то знаете, что код был считан правильно.

Десять цифр кода обычной этикетки со штрихкодом предоставляют возможность записать в него только идентификационный номер товара. Из-за этого любое приложение, занимающееся обработкой штрихкодов, должно использовать базу данных, соотносящую считанный штрихкод с соответствующим товаром. В течение долгого времени заветной целью мира штрихкодов была разработка более емкой штрихкодовой символики, позволяющей кодировать целые документы и обеспечивающей их надежное воспроизведение.

— PDF-417 является нашей новой двумерной штрихкодовой технологией, — объяснил Инджиун. Возможно, вам больше знакомы простые коды QR, но в настоящее время общепринятым стандартом является именно стандарт PDF-417. (Пример этикетки со штрихкодом этого типа показан на рис. 10.7.) Между прочим, обратная сторона каждого водительского удостоверения штата Нью-Йорк содержит всю историю конфликтов с законом его собственника, закодированную в PDF-417.

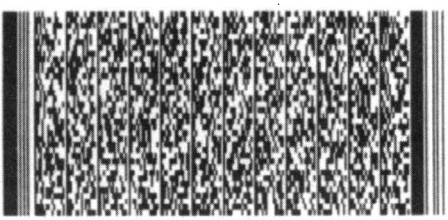


**Рис. 10.7.** Этикетка с Геттисбергским посланием, закодированным двумерным штрихкодом по технологии PDF-417

— Какой объем информации можно уместить с помощью этой технологии на одном квадратном дюйме? — спросил я у него.

— Это зависит от заданного уровня коррекции ошибок, но в общем около 1000 байтов, что достаточно для небольшого текстового файла или изображения, — ответил он.

— Вам, наверное, приходится использовать какую-либо технологию сжатия данных, чтобы максимизировать объем сохраняемого на этикетке текста. (Стандартные алгоритмы сжатия данных рассматриваются в *разд. 21.5*.)

— Да, мы действительно используем определенный способ сжатия данных, — согласился Инджиун. — Мы знаем, какие типы текста наши клиенты станут помещать на этикетки. Некоторые тексты будут состоять полностью из прописных букв, а другие — из букв обоих регистров и цифр. Наш код предоставляет три разных текстовых режима, каждый из которых поддерживает отдельное подмножество алфавитно-цифровых знаков. Мы можем описать каждый знак, используя только пять битов, при условии, что режим не меняется. Для описания знака из другого режима мы сначала выдаем команду переключения режимов (длиной в пять битов), а потом код символа. (Схема переключения режимов для разных групп символов показана на рис. 10.8.)

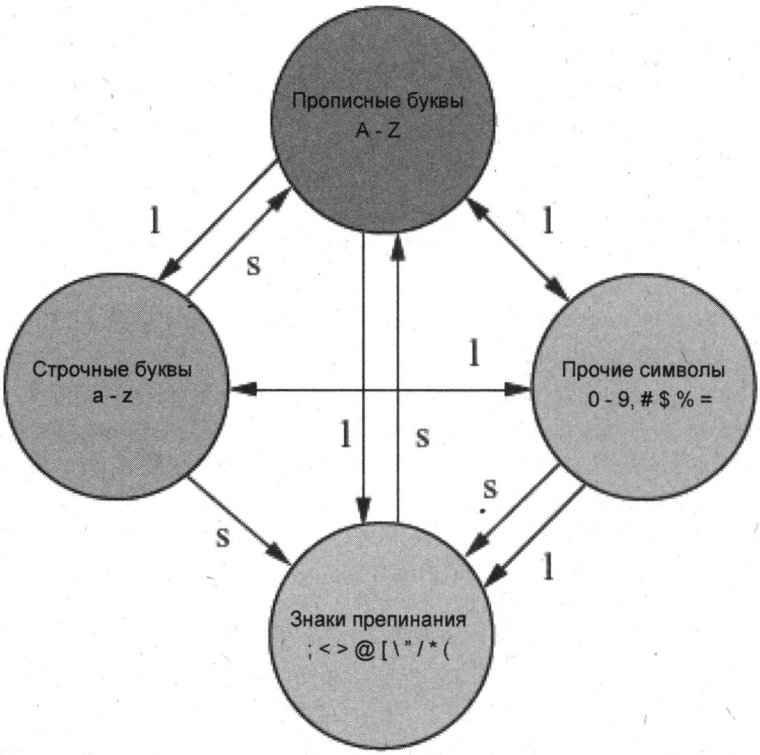



**Рис. 10.8.** Схема переключения режимов в PDF-417

— Понятно. Значит, вы определили для каждого режима группы символов таким образом, чтобы свести к минимуму операции переключения режимов при работе с типичными текстовыми файлами.

— Совершенно верно. Мы поместили все цифры в один режим, а все символы знаков препинания в другой. Мы также реализовали команды *переключения* (switch) и *фиксирования* (latch) режимов. Операция переключения режимов применяется для переключения в новый режим только для следующего символа — скажем, знака препинания. А операция фиксирования режимов используется для постоянного переключения в новый режим, если в этом режиме нужно вывести несколько символов подряд, — например, номер телефона.

— Интересно! — сказал я. — Со всеми этими переключениями режимов должно существовать множество разных способов закодировать текст в виде штрихкода. Как вы получаете закодированный текст наименьшего объема?

— Мы используем «жадный» алгоритм, который выполняет просмотр на несколько символов вперед и решает, какой режим лучше всего использовать. Этот способ работает достаточно хорошо.

Я продолжал выпытывать у него подробности по этому вопросу: «Откуда вы знаете, что этот способ работает хорошо? Ведь может существовать значительно лучшая кодировка, которую вы просто не находите».

— Вообще говоря, мы этого не знаем. Но поиск оптимальной кодировки, вероятно, является NP-полной задачей. Не так ли? — спросил он упавшим голосом.

Я задумался. Каждая кодировка начинается в определенном режиме и состоит из последовательности кодов символов и операций переключения и фиксирования режимов. С любой позиции в тексте мы можем помещать код следующего символа (если он имеется в текущем режиме) или выполнять операцию переключения или фиксирования режима. При продвижении в тексте слева направо текущее состояние полностью отражается текущей позицией символа и текущим состоянием режима. Для заданной пары «позиция/режим» нас интересует самая дешевая из всех возможных кодировок достижения этой точки...

Мои глаза загорелись так ярко, что стали отбрасывать на стены тени от окружающих предметов.

— Оптимальную кодировку для любого текста в PDF-417 можно определить с помощью динамического программирования. Для каждого возможного режима $1 \le m \le 4$ и для каждой возможной позиции символа $1 \le i \le n$ мы сохраняем в матрице $M[i, m]$ информацию о стоимости самой дешевой кодировки первых $i$ символов, заканчивающейся в режиме $m$. Нашим следующим ходом из каждой пары состояний «позиция/режим» является или сопоставление, или переключение, или фиксирование режима, так что рассмотрению подлежит очень небольшое число возможных операций в каждой позиции.

По сути, мы имеем следующее соотношение:

$$M[i, j] = \min_{1 \le m \le 4} \left( M[i-1, m] + c(S_i.m, j) \right),$$

где $c(S_i, m, j)$ представляет стоимость кодирования символа $S_i$ и переключения из режима $m$ в режим $j$. Самая дешевая возможная кодировка получается в результате обратной трассировки от узла $M[n, m]$, где $m$ представляет значение $k$, которое минимизирует $M[n, k]$. Каждую из $4n$ ячеек можно заполнить за постоянное время, так что оптимальную кодировку можно найти за линейное по отношению к длине строки время.

Инджиун отнесся к этому несколько скептически, но попросил, чтобы мы реализовали для него оптимальный метод кодирования. У нас возникли определенные затруднения в связи с некоторыми странностями переключения режимов, но мой студент Йо-Линг Лин (Yaw-Ling Lin) оказался на высоте и решил эту задачу. В компании Symbol Technologies сравнили наш кодировщик со своим, обработав для этого 13 000 этикеток, и пришли к выводу, что динамическое программирование в среднем позволяло поместить на этикетку на 8% больше информации. Это было важным улучшением, поскольку никому не хочется терять 8% емкости хранения, особенно в ситуации, когда максимальная емкость хранения составляет всего лишь несколько сот байтов. Конечно же, *среднее* улучшение на 8% означало, что для определенных типов информации улучшение было намного большим, при этом наш кодировщик никогда не использовал больше памяти, чем их. И хотя наш кодировщик работал чуть медленнее, чем кодировщик на базе «жадного» алгоритма, это не меняло сути достижения, поскольку в любом случае узким местом оставалась распечатка этикеток.

Наблюдаемый нами эффект замены эвристического алгоритма алгоритмом поиска глобального оптимального решения, пожалуй, является типичным для большинства при-

ложений. В общем, если вы не допустите серьезных ошибок в реализации эвристического алгоритма, то вы должны получить приемлемое решение. Но использование вместо него алгоритма поиска оптимального результата обычно дает хоть и небольшое, но заметное улучшение, которое может иметь положительный эффект для вашего приложения.

## 10.5. Неупорядоченное разбиение или сумма подмножества

В задаче о *рюкзаке*, или *сумме подмножеств*, требуется установить, существует ли такое подмножество $S'$ входного мультимножества из $n$ натуральных чисел $S = \{s_1, ..., s_n\}$, сумма членов которого равнялась бы целевому значению $k$. Представьте себе туриста, который хочет полностью заполнить свой рюкзак емкостью в $k$ вещей возможными подмножествами вещей из общего множества вещей $S$. Применения этой важной задачи рассматриваются более подробно в *разд. 16.10*.

Динамическое программирование лучше всего работает с линейно упорядоченными элементами, поэтому мы будем рассматривать элементы слева направо. Упорядочение элементов в мультимножестве $S$ от $s_1$ до $s_n$ делает это возможным. Чтобы сформулировать рекуррентное соотношение, нам нужно выяснить, какая требуется информация по элементам $s_1...s_{n-1}$, чтобы решить, что делать с элементом $s_n$.

Идея состоит в следующем. Или $n$-е натуральное число $s_n$ входит в подмножество, сумма членов которого составляет $k$, или не входит. Если входит, тогда должно быть возможным создать подмножество из первых $n-1$ элементов мультимножества $S$, сумма членов которого составляет $k - s_n$, в результате чего добавление этого последнего элемента может создать требуемую сумму $k$. Если же не входит, тогда может существовать решение, не использующее элемент $s_n$. Совместно эти условия определяют следующее рекуррентное соотношение:

$$T_{n,k} = T_{n-1,k} \vee T_{n-1,k-s_n}.$$

На этом основании можно создать следующий алгоритм с временной сложностью $O(nk)$ для определения, реализуема ли целевая сумма $k$ (листинг 10.16).

**Листинг 10.16. Алгоритм для определения возможности получения суммы *k***

```
bool sum[MAXN+1][MAXSUM+1];     /* Таблица реализуемых сумм */
int parent[MAXN+1][MAXSUM+1];   /* Таблица указателей на родительские
                                   элементы */
bool subset_sum(int s[], int n, int k) {
int i, j; /* Счетчики*/
   sum[0][0] = true;
   parent[0][0] = NIL;
   for (i = 1; i <= k; i++) {
      sum[0][i] = false;
      parent[0][i] = NIL;
   }
```

```
   for (i = 1; i <= n; i++) { /* Создаем таблицу */
      for (j = 0; j <= k; j++) {
          sum[i][j] = sum[i-1][j];
          parent[i][j] = NIL;
          if ((j >= s[i-1]) && (sum[i-1][j-s[i-1]]==true)) {
             sum[i][j] = true;
             parent[i][j] = j-s[i-1];
          }
      }
   }
   return(sum[n][k]);
}
```

В таблице указателей на родительские элементы закодирована информация о фактическом подмножестве чисел, сумма которых составляет $k$. Соответствующее подмножество существует во всех случаях, когда `sum[n][k]==true`, но оно не содержит элемент $s_n$, когда `parent[n][k]==NIL`. Вместо этого мы выполняем обход матрицы, пока не найдем представляющий интерес родительский элемент, а затем следуем по соответствующему указателю, как показано в листинге 10.17.

**Листинг 10.17. Поиск подходящего родительского элемента**

```
void report_subset(int n, int k) {
   if (k == 0) {
        return;
   }
   if (parent[n][k] == NIL) {
        report_subset(n-1,k);
   }
   else {
        report_subset(n-1,parent[n][k]);
          printf(" %d ",k-parent[n][k]);
   }
}
```

На рис. 10.9 приведена таблица `sum` для входного множества $S = \{1, 2, 4, 8\}$ и целевой суммы $k = 11$. Метка $T$ (true, истина) в нижней правой ячейке означает, что целевая сумма может быть получена. Поскольку в рассматриваемом случае все члены входного множества $S$ являются степенями двойки и все целевые натуральные числа можно представить в двоичном виде, то все ячейки нижнего ряда содержат метку $T$.

| $i$ | $s_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | T | F | F | F | F | F | F | F | F | F | F | F |
| 1 | 1 | T | T | F | F | F | F | F | F | F | F | F | F |
| 2 | 2 | T | T | T | T | F | F | F | F | F | F | F | F |
| 3 | 4 | T | T | T | T | T | T | T | T | F | F | F | F |
| 4 | 8 | T | T | T | T | T | T | T | T | T | T | T | T |

**Рис. 10.9.** Таблица `sum` для входного множества $S = \{1, 2, 4, 8\}$ и целевой суммы $k = 11$

А таблица, представленная на рис. ЦВ-10.10, содержит соответствующий массив *родительских* элементов, кодирующий решение 1 + 2 + 8 = 11. Значение 3 в правой нижней ячейке отображает тот факт, что 11 – 8 = 3.

Внимательный читатель может задаться вопросом, как алгоритм для получения суммы подмножества может иметь время исполнения $O(nk)$, когда задача нахождения суммы подмножества является NP-полной? Разве это выражение не многочлен относительно $n$ и $k$? Неужели мы только что доказали, что $P = NP$?

К сожалению, нет. Обратите внимание на то, что целевое число $k$ можно задать, используя $O(\log k)$ битов, — иными словами, время исполнения этого алгоритма экспоненциально размеру его ввода, который составляет $O(n \log k)$. По этой же причине время разложения на множители целого числа $N$ с явной проверкой всех $\sqrt{N}$ кандидатов на наименьший множитель неполиномиальное, поскольку время исполнения экспоненциально $O(\log N)$ битов ввода.

На задачу можно посмотреть с другой стороны, рассмотрев, что происходит с алгоритмом при умножении на 1 000 000 каждого целого числа определенного экземпляра задачи. Такое преобразование не повлияло бы на время исполнения алгоритма сортировки, или поиска минимального остовного дерева, или любого другого алгоритма, рассмотренного нами на этом этапе. Но оно замедлило бы наш алгоритм динамического программирования в 1 000 000 раз, а также потребовался бы в миллион раз больший объем памяти для хранения таблицы. В задаче суммы подмножества диапазон чисел имеет значение, и для больших целых чисел задача становится NP-сложной.

## 10.6. История из жизни: Баланс мощностей

Одно из моих многих (возможно, слишком многих) недобрых подозрений — большинство современных студентов в области электротехники (ЭТ) не знают, как собрать радиоприемник. Причина этого в том, что те студенты ЭТ, с которыми мне приходится работать, изучают одновременно как электротехнику, так и *вычислительную* технику, концентрируясь при этом на архитектуре компьютеров и встроенных систем, что в равной степени включает программное и аппаратное обеспечение. Боюсь, что в случае стихийного бедствия эти молодые люди будут не в состоянии восстановить работу моей любимой AM-радиостанции.

Поэтому было облегчением узнать, что еще не все потеряно, когда профессор кафедры электротехники со своими студентами обратились ко мне в поисках помощи с оптимизацией работы электрической сети.

— В энергетических системах переменного тока электричество передается по трем разным фазам. Назовем их $A$, $B$ и $C$. Система работает наиболее эффективно, когда нагрузка на всех фазах примерно одинаковая, — объяснил профессор.

— Я полагаю, что под нагрузкой имеется в виду оборудование, требующее питания, правильно? — догадался я.

— Да, каждый дом на улице можно рассматривать как нагрузку. Каждый дом подключается к одной и трех фаз, играющей роль его источника электроэнергии.

— Предположительно, чтобы сбалансировать нагрузку на линию, дома подключаются к фазам поочередно в порядке *A*, *B*, *C*, *A*, *B*, *C* и т. д.?

— Что-то типа этого, — подтвердил профессор. — Но не все дома потребляют одинаковый объем электроэнергии, а еще хуже дело обстоит в индустриальных районах. Одна компания может лишь включать освещение, тогда как другая подключает дуговую печь. После замеров фактического использования электроэнергии каждым потребителем мы хотим переключить некоторых из них на другие фазы, чтобы сбалансировать фазную нагрузку.

Здесь я увидел алгоритмическую задачу: «То есть для имеющегося множества значений, представляющих разные нагрузки, вы хотите сопоставить их фазам *A*, *B* и *C* таким образом, чтобы сбалансировать нагрузку как можно более равномерно, правильно?»

— Да. Можете ли вы дать мне быстрый алгоритм для этого? — спросил он.

Задача казалась мне вполне понятной. Это выглядело как задача разделения множества целых чисел, — в частности, напоминало задачу суммы подмножества, рассмотренную в *разд 10.5*, где целевая сумма $k = \left(\sum_{i=1}^{n} s_n\right)/2$. Наиболее сбалансированное разбиение получается тогда, когда сумма элементов выбранного подмножества (в нашем случае $k$) равняется сумме оставшихся элементов (в нашем случае $\sum_{i=1}^{n} s_n - k$).

Напрашивалось расширение задачи до разбиения на три подмножества вместо двух, но это нисколько не делало ее решение более легким. Добавление одного нового элемента $s_{n+1} = k$ и необходимость разбиения множества $S$ на три подмножества равного веса требует решения задачи разделения множества целых чисел с исходными элементами.

Я попытался сообщить им плохую новость как можно деликатнее: «Задача разбиения множества целых чисел является NP-полной, и задача балансировки нагрузки на три фазы настолько трудная, насколько она только может быть. Алгоритма с полиномиальным временем исполнения для решения вашей задачи не существует».

Они поднялись и направились к выходу. Но тут я вспомнил об алгоритме динамического программирования для решения задачи суммы элементов подмножества, рассмотренной в *разд. 10.5*, и остановил их. Почему этот алгоритм нельзя расширить на три подмножества — т. е. фазы? В частности, определяем функцию $C[n, A, B]$ для имеющегося множества нагрузок $S$, где $C[n, w_A, w_B]$ возвращает истину, если существует способ разбить первые $n$ нагрузок множества $S$ таким образом, чтобы вес фазы $A$ был $w_A$, а фазы $B$ — $w_B$. Следует также обратить внимание на отсутствие необходимости явно отслеживать вес фазы $C$, поскольку $w_C = \sum_{i=1}^{n} s_i - w_A - w_B$. Тогда мы получим следующее рекуррентное соотношение, определяемое подмножеством, на которое устанавливается $n$-я нагрузка:

$$C[n, w_A, w_B] = C[n-1, w_A - s_n, w_B] \vee C[n-1, w_A, w_B - s_n] \vee C[n-1, w_A, w_B].$$

Обновление этого выражения занимало постоянное время для каждой ячейки, а обновлению подлежало $nk^2$ ячеек, где $k$ представляет максимальное значение мощности для любой фазы, с которым мы были готовы работать. Таким образом, мы могли бы оптимально сбалансировать нагрузку за $O(nk^2)$ время.

Мои посетители были чрезвычайно рады такому решению и направились к выходу, чтобы не мешкать с реализацией алгоритма. Но прежде, чем они ушли, у меня был один вопрос, который я целенаправленно задал одному из студентов в области вычислительной техники: «Почему для переменного тока используются три фазы?»

— Мммм... возможно, что-то связано с согласованием полных сопротивлений и, мммм... с комплексными числами? — промычал он.

Профессор косо посмотрел на него, а я почувствовал удовлетворение о того, что все-таки мои недобрые подозрения не были безосновательными. Однако этот студент, обучающийся основам вычислительной техники, умел разрабатывать программы, что было решающим аспектом в нашем случае. Он быстро реализовал предложенный алгоритм динамического программирования и провел эксперименты на представительных задачах, результаты которых излагаются в [WSR13].

Решение, выдаваемое нашим алгоритмом динамического программирования, всегда было таким же качественным, как и решения, выдаваемые несколькими эвристическими алгоритмами, и обычно даже лучшим. В этом не было ничего удивительного, поскольку наш алгоритм всегда выдавал оптимальное решение, а другие алгоритмы — нет. Время исполнения нашей динамической программы возрастало квадратически в диапазоне нагрузок, что могло представлять проблему. Но разбиение нагрузок на интервалы величиной, скажем, $\lfloor s_i / 10 \rfloor$ уменьшило бы время исполнения в 100 раз и позволило бы получать решения, которые все еще были достаточно хорошими для исходной задачи.

Динамическое программирование по-настоящему доказало свою полезность, когда наши электротехники обратили свой интерес на более амбициозные целевые функции. Операция переноса нагрузок с одной фазы на другую имела свою стоимость, поэтому они хотели вычислить относительно сбалансированное распределение нагрузок по фазам, которое бы минимизировало количество изменений, требуемых для его получения. Эта задача решается, по сути, таким же рекуррентным соотношением при сохранении самой низкой стоимости реализации каждого состояния вместо простой установки флага, указывающего возможность его получения:

$$\begin{aligned} C[n, w_A, w_B] = \min & C[n-1, w_A - s_n, w_B] + 1, \\ & C[n-1, w_A, w_B - s_n] + 1, \\ & C[n-1, w_A, w_B]. \end{aligned}$$

А затем они разошлись вовсю и захотели получить решение с самой низкой стоимостью, которое никогда по-серьезному не разбалансировалось бы в любой точке на линии. В глобально сбалансированном решении можно выбрать заполнение общего объема нагрузки на фазе $A$ до того, как подключать какие-либо нагрузки на фазу $B$ или $C$, и это плохо. Но то же самое рекуррентное соотношение все равно справится с поставленной задачей при условии, что задается $C[n, w_A, w_B] = \infty$ всякий раз, когда нагрузки в этом состоянии считаются слишком несбалансированными, чтобы быть приемлемыми.

В этом и заключается мощь динамического программирования. Достаточно уменьшить пространство состояний до приемлемого небольшого размера, чтобы можно было

оптимизировать практически что угодно. Для этого просто проходим через каждое возможное состояние и присваиваем ему соответствующую стоимость.

## 10.7. Задача упорядоченного разбиения

Допустим, что нужно просмотреть несколько книг на полке, чтобы найти определенную информацию. Для выполнения этого задания выделено трое сотрудников, каждому из которых будет дана определенная часть книг для просмотра. Чтобы сохранить начальный порядок книг, проще всего разделить полку на три части и назначить по части каждому сотруднику для просмотра.

Но как правильно распределить книги среди сотрудников? Если все книги имеют одинаковое количество страниц, то тогда это сделать очень легко: разделить все книги на три равные части. Например:

100 100 100 | 100 100 100 | 100 100 100.

Таким образом, каждому сотруднику достанутся три книги с общим количеством в триста страниц.

А если количество страниц в книгах неодинаковое? Допустим, что мы распределили таким же образом следующие книги:

100 200 300 | 400 500 600 | 700 800 900.

Я лично предпочел бы просматривать первую часть размером только в 600 страниц, но не последнюю размером в 2400 страниц. В этом случае самое справедливое распределение книг было бы следующим:

100 200 300 400 500 | 600 700 | 800 900.

То есть самая большая часть будет содержать 1700 страниц.

В общем виде задачу можно сформулировать так:

**Задача**. Разделение множества целых чисел на подмножества без перестановок.

**Вход**. Множество $S$ положительных чисел $s_1$, ..., $s_n$ и целое число $k$.

**Выход**. Разделить множество $S$ на $k$ или меньше подмножеств таким образом, чтобы максимальная из сумм подмножеств была как можно меньше.

Такая задача, называемая задачей *упорядоченного разбиения* (ordered partition), часто возникает в параллельных процессах, когда нужно распределить нагрузку среди нескольких процессоров таким образом, чтобы минимизировать общее время исполнения.

Узким местом в такого рода вычислениях является процессор, которому назначено больше всего работы. Кстати, в истории из жизни в *разд. 5.8* было описано неудачное решение именно такой задачи.

Отложите книгу на несколько минут и попробуйте разработать алгоритм для решения задачи упорядоченного разбиения.

Для начинающего алгоритмиста наиболее естественным подходом к решению этой задачи может казаться использование эвристического алгоритма. Например, он может вычислить средний размер раздела, равный $\sum_{i=1}^{n} s_i / k$, после чего попытаться выпол-

нить разбиение как можно ближе к этому среднему. Но такие эвристические методы решения обречены на провал при некоторых входных экземплярах, поскольку они не выполняют систематическую оценку всех возможных решений.

Вместо этого для решения этой задачи рассмотрим рекурсивный подход исчерпывающего поиска. Обратите внимание на то, что $k$-й раздел начинается сразу же после $(k-1)$-го разделителя. Где же мы можем поставить этот последний разделитель? Между $i$-м и $(i+1)$-м элементами для $i$ такого, что $1 \le i \le n$. Каковы будут наши затраты после этой вставки? Общей стоимостью будет большая из двух величин:

- стоимость последнего раздела $\sum_{j=i+1}^{n} s_j$ или
- стоимость наибольшего раздела, созданного слева от последнего разделителя.

Каким будет размер этого левого раздела? Чтобы минимизировать общую стоимость, нам нужно разделить $s_1$, ..., $s_j$ элементов по возможности на одинаковые части, используя оставшиеся $k-2$ разделителей. *Но ведь это та же самая задача, только в меньшем экземпляре, и, значит, ее можно решить рекурсивно!*

Исходя из этого, определим $M[n, k]$ как минимально возможную стоимость по всем операциям разбиения множества $\{s_1, ..., s_n\}$ на $k$ подмножеств, где стоимость разбиения определяется как наибольшая сумма элементов в одной из ее частей. Эту функцию можно вычислить:

$$M[n,k] = \min_{i=1}^{n}\left(\max\left(M[i,k-1], \sum_{j=i+1}^{n} s_j\right)\right).$$

Для рекуррентного соотношения также необходимо указать граничные условия. Эти граничные условия задают наименьшие возможные значения для всех аргументов. Для нашей задачи наименьшим значением первого аргумента будет $n = 1$, а это означает, что первый раздел состоит из одного элемента. Независимо от количества применяемых разделителей, первый раздел нельзя создать меньшим чем $s_1$. Наименьшим значением второго аргумента будет $k = 1$, а это означает, что множество $S$ вообще не разбивается на подмножества. Таким образом:

$$M[1, k] = s_1 \text{ для всех } k > 0 \text{ и}$$

$$M[n,1] = \sum_{i=1}^{n} s_i.$$

Сколько времени уйдет на вычисление, если сохранять частичные результаты? Всего в таблице имеется $k \cdot n$ ячеек. Сколько времени займет вычисление значений $M[n', k']$ для $1 \le n' \le n$, $1 \le k' \le k$? Для вычисления этой величины при помощи обобщенного рекуррентного соотношения требуется найти минимальную из $n'$ величин, каждая из которых является большим из двух значений: значения в таблице поиска и суммы самое большее $n'$ элементов (занимая время $O(n')$). Если заполнение каждой из $kn$ ячеек занимает время $n^2$, то всю рекуррентность можно вычислить за время $O(kn^3)$.

Меньшие значения вычисляются раньше больших с тем, чтобы на каждом шаге вычислений имелись необходимые данные. Реализация алгоритма приводится в листинге 10.18.

**Листинг 10.18. Реализация алгоритма решения задачи линейного разбиения**

```
void partition(int s[], int n, int k) {
    int p[MAXN+1];              /* Массив префиксных сумм */
    int m[MAXN+1][MAXK+1];      /* Таблица значений */
    int d[MAXN+1][MAXK+1];      /* Таблица разделителей */
    int cost;                   /* Стоимость тестового разбиения */
    int i,j,x;                  /* Счетчики */

    p[0] = 0;                   /* Создаем префиксные суммы */
    for (i = 1; i <= n; i++) {
        p[i] = p[i-1] + s[i];
    }

    for (i = 1; i <= n; i++) {
        m[i][1] = p[i];           /* Инициализируем границы */
    }

    for (j = 1; j <= k; j++) {
        m[1][j] = s[1];
    }

    for (i = 2; i <= n; i++) {     /* Вычисляем основное рекуррентное
                                      соотношение */
        for (j = 2; j <= k; j++) {
            m[i][j] = MAXINT;
            for (x = 1; x <= (i-1); x++) {
                cost = max(m[x][j-1], p[i]-p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }
    }
    reconstruct_partition(s, d, n, k); /* Выводим решение разделения */
}
```

В действительности реализация из листинга 10.18 работает быстрее, чем мы рассчитывали. При первоначальном анализе предполагалось, что обновление каждой ячейки матрицы занимает время $O(n^2)$. Это предположение было основано на том обстоятельстве, что мы выбираем наилучшую из самое большее $n$ возможных точек для размещения разделителя, для каждого из которых требуется сумма из самое большее $n$ возможных членов. Но в действительности можно с легкостью избежать вычисления этих сумм, сохраняя набор из $n$ префиксных сумм:

$$p_i = \sum_{k=1}^{i} s_k,$$

поскольку

$$\sum_{k=i}^{j} s_k = p_j - p_{i-1}.$$

Это позволяет вычислять каждую ячейку за линейное время, что дает время исполнения $O(kn^2)$. Указанные префиксные суммы также используются в качестве инициализирующих значений для $k = 1$ и показаны в матрицах динамического программирования (рис. ЦВ-10.11).

Изучив рекуррентное соотношение и матрицы динамического программирования, представленные на рис. ЦВ-10.11, вы должны убедиться в том, что конечное значение $M[n, k]$ будет стоимостью наибольшего диапазона в оптимальном разбиении.

Но для большинства приложений нам требуется информация о том, как фактически выполнять само разбиение. Без этой информации мы, образно выражаясь, имеем лишь купон с большой скидкой на товар, которого нет на складе.

Для восстановления оптимального решения используется вторая матрица $D$. При каждом обновлении значения $M[i, j]$ мы записываем позицию разделителя, которая использовалась для получения этого значения. Чтобы восстановить путь, который привел к оптимальному решению, мы идем назад от $D[n, k]$ и добавляем разделитель в каждой указанной позиции. Эту обратную трассировку лучше всего выполнять с помощью рекурсивной процедуры, приведенной в листинге 10.19.

**Листинг 10.19. Рекурсивная процедура восстановления решения**

```
void reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k) {
    if (k == 1) {
        print_books(s, 1, n);
    } else {
        reconstruct_partition(s, d, d[n][k], k-1);
        print_books(s, d[n][k]+1, n);
    }
}

void print_books(int s[], int start, int end) {
    int i; /* counter */

    printf("\{");
    for (i = start; i <= end; i++) {
       printf(" %d ", s[i]);
    }
    printf("}\n");
}
```

# 10.8. Синтаксический разбор

При компилировании исходного кода программы компилятор определяет, отвечает ли структура программы требованиям синтаксиса соответствующего языка программирования. В случае полного соответствия программа компилируется, в противном случае компилятор выдает сообщение об ошибке. Для этого требуется точное описание синтаксиса языка, которое обычно дается *контекстно-свободной грамматикой*, как показано в примере на рис. 10.12, *а*.

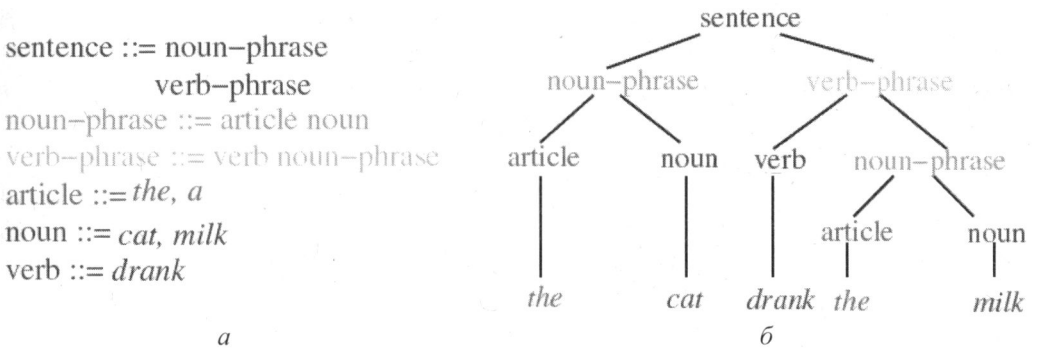



**Рис. 10.12.** Контекстно-свободная грамматика (*а*) и соответствующее дерево синтаксического разбора (*б*)

Каждое *правило* (production) грамматики определяет интерпретацию именованного символа в левой части правила в виде последовательности символов в правой части правила. Правая сторона правила может содержать сочетание нетерминальных символов, или просто *нетерминалов* (которые также определяются правилами), или *терминальных* символов, которые определяются просто как строки, — например: `the`, `a`, `cat`, `milk` и `drank`.

*Синтаксический разбор* (parsing) текстовой последовательности $S$ согласно правилам контекстно-свободной грамматики $G$ представляет собой алгоритмическую задачу создания дерева синтаксического разбора правил замены, определяющего строку $S$ в виде единого нетерминального символа грамматики $G$. На рис. 10.12, *б* показано дерево синтаксического разбора простого предложения согласно правилам грамматики, представленным на рис. 10.12, *а*.

В мои студенческие времена синтаксический разбор казался мне ужасно сложным предметом. Но недавно один приятель без труда объяснил мне его за ланчем. Правда, теперь я понимаю динамическое программирование намного лучше, чем тогда, когда был студентом.

Мы полагаем, что длина последовательности $S$ равна $n$ символам, а сама грамматика $G$ имеет постоянный размер. Такое предположение справедливо, поскольку независимо от размера компилируемой программы грамматика определенного языка программирования (например, C или Java) имеет фиксированный размер.

Кроме этого, мы полагаем, что определения каждого правила представлены в *нормальной форме Хомского* (Chomsky normal form), как в примере, показанном на рис. 10.12. Это означает, что правая сторона каждого правила состоит или ровно из двух нетерминальных символов — например: $X \rightarrow Y\,Z$, или ровно из одного терминального символа — например: $X \rightarrow \alpha$. Любую контекстно-свободную грамматику можно механически преобразовать в нормальную форму Хомского, многократно сокращая длинные правые стороны за счет добавления дополнительных нетерминальных символов и правил. Таким образом, сделанное предположение не приводит к потери общности.

Как можно выполнить эффективный синтаксический разбор текстовой последовательности $S$, используя контекстно-свободную грамматику, в которой каждое представляющее интерес правило создает два нетерминальных символа? Здесь ключевым

является то обстоятельство, что правило, применяемое в корне дерева синтаксического разбора (скажем, $X \rightarrow Y\,Z$), разделяет строку $S$ в позиции $i$ таким образом, что левая часть строки $(S_1 \ldots S_i)$ *генерируется* нетерминальным символом $Y$, а правая часть $(S_{i+1} \ldots S_n)$ — элементом $Z$.

Это обстоятельство наводит на мысль об использовании динамического программирования, в котором мы отслеживаем все нетерминальные символы, генерируемые каждой непрерывной подпоследовательностью текстовой последовательности $S$. Определяем булеву функцию $M[i, j, X]$, которая возвращает значение *истина* тогда и только тогда, когда подстрока $S_i \ldots S_j$ генерируется нетерминальным символом $X$. Это происходит, когда существует правило $X \rightarrow Y\,Z$ и такая точка разрыва $k$ между $i$ и $j$, что левая часть генерирует $Y$, а правая — $Z$. Иными словами, для $i < j$ имеем

$$M[i,j,X] = \bigvee_{(X \rightarrow Y\,Z) \in G} \left( \bigvee_{k=i}^{j-1} M[i,k,Y] \wedge M[k+1,j,Z] \right),$$

где символ $\bigvee$ означает логическое ИЛИ по всем правилам и позициям разделения, а символ $\wedge$ — операцию логического И над двумя булевыми значениями.

Терминальные символы определяют граничные условия рекуррентности. В частности, $M[i, i, X]$ определяется как *истина* тогда и только тогда, когда существует такое правило $X \rightarrow \alpha$, что $S_i = \alpha$.

Какова временная сложность этого алгоритма? Размер пространства состояний равен $O(n^2)$, поскольку существует $n(n+1)/2$ подпоследовательностей, определяемых парами $(i, j)$, где $i \geq j$. Умножение этого значения на количество нетерминальных символов, которое является конечным, т. к. грамматика определена постоянным размером, не влияет на асимптотический верхний предел. Чтобы вычислить значение $M[i, j, X]$, нужно протестировать все промежуточные значения $k$, где $i \leq k < j$, так что вычисление каждой из $O(n^2)$ ячеек в наихудшем случае занимает время $O(n)$. Отсюда получаем кубическую — т. е. $O(n^3)$ — временную сложность алгоритма синтаксического разбора.

## Остановка для размышлений: Экономичный синтаксический разбор

**ЗАДАЧА**. Нередко в программах имеются синтаксические ошибки, из-за которых она не компилируется. Для заданной контекстно-свободной грамматики $G$ и последовательности ввода $S$ определить наименьшее количество замен символов, которые нужно выполнить в последовательности $S$, чтобы конечная последовательность была приемлемой для грамматики $G$.

**РЕШЕНИЕ.** Когда я впервые столкнулся с этой задачей, она казалась чрезвычайно трудной. Но после некоторых размышлений я пришел к выводу, что она является просто очень общим случаем задачи вычисления расстояния редактирования, для решения которой динамическое программирование подходит самым естественным образом. Синтаксический разбор также вначале казался трудной задачей, но потом поддался решению тем же методом. Действительно, мы можем решить объединенную задачу, обобщив на нее отношение рекуррентности, которое мы использовали для выполнения простого синтаксического разбора.

Определим *целочисленную функцию* $M'[i, j, X]$, которая возвращает количество минимальных изменений в подпоследовательности $Si \ldots Sj$, позволяющих создать ее не-

терминальным символом $X$. Этот символ будет создаваться неким правилом $X \rightarrow Y\,Z$. Некоторые изменения последовательности $S$ могут находиться слева от точки раздела, а другие — справа, но нас интересует только минимизация суммы. Иными словами, для $i < j$ имеем

$$M'[i,j,X] = \min_{(X \rightarrow Y\,Z) \in G} \left( \min_{k=i}^{j-1} M'[i,k,Y] + M'[k+1,j,Z] \right).$$

Также слегка изменяются граничные условия. Если существует правило $X \rightarrow \alpha$, то стоимость сравнения в позиции $i$ зависит от содержимого $S_i$. Если $S_i = \alpha$, тогда $M'[i, i, X] = 0$. В противном случае для изменения $S_i$ на $\alpha$ можно расплатиться одной заменой, поэтому $M'[i, i, X] = 1$, если $S_i \neq a$. Но если в грамматике отсутствует правило $X \rightarrow \alpha$, то не существует способа выполнить замену в односимвольной строке, чтобы получить что-либо, создающее $X$, поэтому $M'[i, i, X] = \infty$ для всех $i$. ■

#### Подведение итогов

Для любой задачи оптимизации объектов, упорядоченных слева направо, например символов строки, элементов перестановки, вершин многоугольника или листьев дерева поиска, применение динамического программирования, скорее всего, позволит создать эффективный алгоритм для получения оптимального решения.

## 10.9. Ограничения динамического программирования: задача коммивояжера

Динамическое программирование подходит не для всех задач. Важно понимать, почему его применение может не дать желаемого результата, и уметь избегать ситуаций, чреватых созданием неправильного или неэффективного алгоритма.

Материалом для наших алгоритмических экспериментов снова будет задача коммивояжера, в которой мы ищем самый короткий маршрут для посещения всех городов. Но мы ограничимся следующим случаем:

**Задача**. Самый длинный простой путь.

**Вход.** Взвешенный граф $G = (V, E)$, в котором указаны начальная ($s$) и конечная ($t$) вершины.

**Выход.** Самый длинный маршрут от $s$ до $t$, в котором все вершины посещаются только один раз.

Между этой задачей и задачей коммивояжера имеются два несущественных различия. Во-первых, требуется найти путь, а не замкнутый маршрут. Эта разница несущественна, поскольку мы можем получить замкнутый маршрут, просто включив в путь ребро $(t, s)$. Во-вторых, требуется найти наиболее длинный путь, а не наиболее короткий маршрут. Опять же, разница не является существенной: нам просто требуется посетить как можно больше вершин (в идеале все), точно так же, как и в задаче коммивояжера. Ключевым словом в постановке этой задачи является слово *простой*, означающее, что любую вершину мы можем посетить только один раз.

Для невзвешенных графов (в которых вес каждого ребра равен единице) вес самого длинного пути от $s$ к $t$ будет равным $n - 1$. Задача поиска таких гамильтоновых путей

(если они существуют) представляет собой важную задачу теории графов, и мы рассмотрим эту задачу в *разд. 19.5*.

## 10.9.1. Вопрос правильности алгоритмов динамического программирования

Алгоритмы динамического программирования правильны лишь настолько, насколько правильны рекуррентные соотношения, на которых они основаны. Допустим, мы определим $LP[i, j]$ как длину максимального простого пути от вершины $i$ к вершине $j$. Обратите внимание на то, что самый длинный простой путь от вершины $i$ к вершине $j$ перед тем, как попасть в вершину $j$, должен пройти через некую вершину $x$. Таким образом, последним ребром пути должно быть ребро $(x, j)$. Здесь напрашивается следующее рекуррентное соотношение для вычисления длины максимального пути, где $c(x, j)$ — вес ребра $(x, j)$:

$$LP[i, j] = \max_{\substack{x \in V \\ (x, j) \in E}} LP[i, x] + c(x, j).$$

Идея кажется разумной, однако я вижу в ней по крайней мере два недостатка.

Прежде всего, в этом рекуррентном соотношении не предусмотрено ничего для обеспечения простоты. Откуда мы знаем, что вершина $j$ не использовалась ранее в самом длинном простом пути от вершины $i$ к вершине $x$? Если использовалась, то тогда добавление ребра $(x, j)$ создаст цикл. Чтобы предотвратить подобное развитие событий, мы должны определить такое рекуррентное соотношение, которое помнит пройденный путь. Возможно, мы могли бы достичь этого, определив $LP'[i, j, k]$ для обозначения самого длинного пути от вершины $i$ к вершине $j$, не включающего вершину $k$? Это был бы шаг в правильном направлении, но он все равно не даст нам работающее должным образом рекуррентное соотношение.

Вторая проблема затрагивает порядок вычисления. Какой элемент мы вычисляем первым? Так как вершины графа не упорядочены слева направо или в порядке возрастания, то неясно, какими должны быть меньшие подпрограммы. При отсутствии такого упорядочивания мы легко окажемся в бесконечном цикле.

Динамическое программирование можно использовать для решения любой задачи, в которой соблюдается *принцип оптимальности*. Иными словами, это означает, что множество частичных решений может быть оптимально расширено на основании *состояний* после частичных решений, а не специфики самих частичных решений. Например, когда мы решали, расширять ли нечеткое сравнение строк за счет замены, вставки или удаления, нам не требовалось знать последовательность операций, выполненную до этого момента. В действительности может существовать несколько разных последовательностей редактирования, которые выдают стоимость $C$ на первых $p$ символах строки образца $P$ и $t$ символах строки $T$. Очередные решения принимаются на основе *последствий* предыдущих решений, а не на основе самих предыдущих решений.

Когда важна не просто стоимость выполняемых операций, а их специфика, задача не удовлетворяет принципу оптимальности. В качестве примера такой задачи можно при-

вести специальную разновидность задачи вычисления расстояния редактирования, в которой не разрешается использовать комбинации операций в определенных последовательностях. Но при правильной постановке задачи принцип оптимальности соблюдается во многих комбинаторных задачах.

## 10.9.2. Эффективность алгоритмов динамического программирования

Время исполнения любого алгоритма динамического программирования зависит от двух факторов: количества частичных решений, которые необходимо отслеживать, и длительности вычисления каждого частичного решения. Обычно более важным является первый аспект — а именно размер пространства состояний.

Во всех приведенных здесь примерах частичные решения полностью описываются посредством указания возможных *точек остановки* при вводе. Это объясняется тем, что элементы обрабатываемых комбинаторных объектов (типично строк и числовых последовательностей) неявно упорядочены. Такое упорядочивание нельзя нарушить, не изменив при этом полностью всю задачу. При установленном порядке элементов существует сравнительно небольшое количество возможных точек остановки или *состояний*, что позволяет получить эффективные алгоритмы.

В случае же отсутствия жесткой упорядоченности объектов количество возможных частичных решений будет, скорее всего, расти экспоненциально. Допустим, что состоянием нашего частичного решения задачи нахождения наидлиннейшего простого пути является весь путь $P$ от начальной до конечной вершины. Таким образом, $LP[i, j, P_{ij}]$ представляет стоимость наидлиннейшего простого пути от вершины $i$ к вершине $j$, где $P_{ij}$ обозначает последовательность промежуточных вершин в этом пути. Это можно правильно вычислить с помощью следующего рекуррентного соотношения, в котором $P + x$ обозначает присоединение вершины $x$ в конец пути $P$:

$$LP\left[i, j, P_{ij}\right] = \max_{\substack{j \notin P_{ix} \\ (x,j) \in E \\ P_{ij} = P_{ix} + j}} LP\left[i, x, P_{ix}\right] + c(x, j).$$

Эта формулировка корректна, но насколько она эффективна? Путь $P_{ij}$ является упорядоченной последовательностью, содержащей вплоть до $n - 3$ вершин, поэтому количество таких путей может доходить до $(n - 3)!$, а это очень много! Фактически этот алгоритм использует комбинаторный поиск (наподобие поиска с возвратом) для создания всех возможных промежуточных путей. Так что в этом случае функция max отчасти вводит в заблуждение, поскольку для создания состояния $LP[i, j, P_{ij}]$ возможно только одно значение $P_{ij}$.

Но этой идее можно найти лучшее применение. Пусть $LP'[i, j, S_{ij}]$ представляет самый длинный простой путь от вершины $i$ к вершине $j$, где *множество* $S_{ij}$ представляет промежуточные вершины в этом пути. Таким образом, если $S_{ij} = \{a, b, c, i, j\}$, то существует ровно шесть путей, совместимых с $S_{ij}$: *iabcj*, *iacbj*, *ibacj*, *ibcaj*, *icabj* и *icbaj*. Это пространство состояний может содержать максимум $2^n$ элементов, что меньше, чем перечисление всех путей. Кроме того, эту функцию можно вычислить с помощью следующего рекуррентного соотношения:

$$LP'\left[i, j, S_{ij}\right] = \max_{\substack{j \notin S_{ix} \\ (x,j) \in E \\ S_{ij} = S_{ix} \cup \{j\}}} LP'\left[i, x, S_{ix}\right] + c(x, j),$$

где $S \cup \{x\}$ обозначает объединение $S$ и $x$.

После этого самый длинный простой путь от вершины $i$ к вершине $j$ можно найти, максимизировав по всем возможным промежуточным подмножествам вершин:

$$LP[i, j] = \max_{S} LP'[i, j, S]$$

Существует только $2^n$ подмножеств множества $n$ вершин, так что это большое улучшение по сравнению с перечислением всех $n!$ маршрутов. На практике этот метод можно использовать для решения задачи коммивояжера с количеством вершин, близким к 30 или даже большим, в то время как значение $n = 20$ будет неприемлемым для алгоритма с временной сложностью $O(n!)$. Тем не менее динамическое программирование оказывается наиболее эффективным на множестве хорошо упорядоченных объектов.

#### Подведение итогов

При отсутствии естественного упорядочивания слева направо алгоритм, использующий динамическое программирование, обычно обречен на экспоненциальную сложность как по времени, так и по памяти.

## 10.10. История из жизни. Динамическое программирование и язык Prolog

— Наш эвристический алгоритм очень хорошо проявляет себя на практике. — Мой коллега одновременно хвастался и просил о помощи.

Унификация является основным вычислительным механизмом в языках логического программирования, таких как Prolog. Программа на языке Prolog состоит из набора правил, где каждое правило имеет голову и действие, которое выполняется, когда голова правила совпадает или объединяется с текущим вычислением.

Выполнение программы на языке Prolog начинается с указания цели — например, $p(a, X, Y)$, где $a$ является константой, а $X$ и $Y$ — переменными. После этого система систематически сравнивает голову цели с головой каждого правила, которое можно унифицировать с целью. Под унификацией имеется в виду привязка переменных к константам, если их можно соотнести. В представленной в листинге 10.20 бессмысленной программе на языке Prolog цель $p(X, Y, a)$ унифицируется с любым из первых двух правил, поскольку для $X$ и $Y$ можно выполнить привязку, чтобы сопоставить дополнительные символы. Цель $p(X, X, a)$ сопоставится только с первым правилом, т. к. привязываемые к первой и второй позиции переменные должны быть одинаковыми.

**Листинг 10.20. Пример программы на языке Prolog**

```
p(a,a,a) := h(a);
p(b,a,a) := h(a) * h(b);
p(c,b,b) :=h(b) + h(c);
p(d,b,b):=h(d) + h(b);
```

— Для ускорения операции унификации мы хотим выполнить предварительную обработку набора правил, чтобы можно было быстро определить, какие правила соответствуют заданной цели. Для этого нам нужно организовать правила в нагруженном дереве (trie), — продолжал мой собеседник.

Нагруженные деревья (см. *разд. 15.3*) являются полезными структурами данных для работы со строками. Каждый лист нагруженного дерева представляет одну строку. Каждый узел на пути от корня к листу маркируется одним символом строки, при этом *i*-й узел пути соответствует *i*-му символу строки.

— Согласен. Нагруженное дерево является естественным способом для представления ваших правил. Создание нагруженного дерева для набора строк символов — задача прямолинейная: мы просто вставляем строки, начиная с корня. Так в чем заключается ваша проблема? — спросил я.

— Эффективность нашего алгоритма унификации очень зависит от минимизирования количества ребер в нагруженном дереве. Так как мы знаем все правила наперед, то у нас имеется свобода действий, чтобы переупорядочить позиции символов в правилах. Например, вместо того, чтобы корневой узел всегда представлял первый аргумент правила, мы можем избрать, чтобы он представлял третий аргумент. Мы бы хотели воспользоваться этой свободой действий для того, чтобы создать нагруженное дерево минимального размера для набора правил.

Мой собеседник показал мне пример (рис. 10.13).

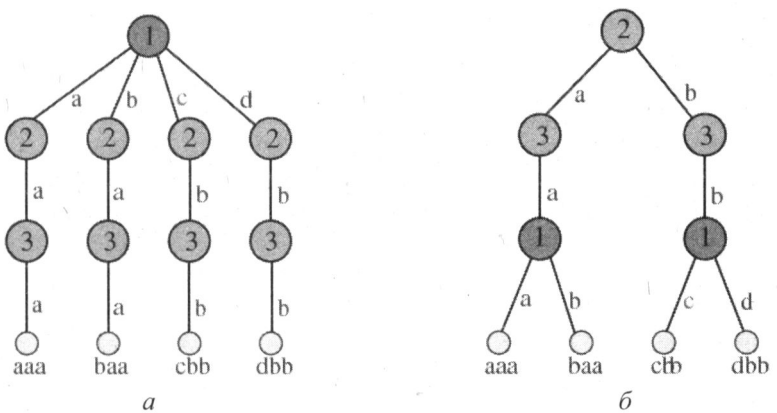



**Рис. 10.13.** Два разных нагруженных дерева для одного и того же набора правил Prolog, где нагруженное дерево (*б*) содержит ребер на четыре меньше

В нагруженном дереве, созданном согласно первоначальному порядку позиций строк (1, 2, 3), всего используется 12 ребер (рис. 10.13, *а*). Но переставив позиции символов в (2, 3, 1) на обеих сторонах, мы могли бы получить нагруженное дерево лишь с 8 ребрами (рис. 10.13, *б*).

— Интересно... — начал было я отвечать, но он снова прервал меня.

— Есть еще одно ограничение. Листья нагруженных деревьев необходимо содержать упорядоченными, чтобы листья основного дерева располагались слева направо в том

же самом порядке, в каком правила выводятся на страницу. Порядок правил в программах Prolog имеет очень, очень большую важность. Если изменить порядок правил, то программа возвратит другой результат.

Потом он перешел к описанию того, что им требовалось от меня.

— У нас есть «жадный» эвристический алгоритм для создания хороших, но не оптимальных, нагруженных деревьев. Этот алгоритм выбирает в качестве корня дерева символ в такой позиции, которая минимизирует степень корня. Иными словами, алгоритм выбирает такую позицию символа, в которой имеется наименьшее число разных символов. Наш эвристический алгоритм работает исключительно хорошо на практике. Но нам нужно доказать, что задача построения наилучшего нагруженного дерева является NP-полной, чтобы наша работа была полностью завершена. Вот в этом нам и нужна ваша помощь.

Я пообещал доказать, что задача имеет такой уровень сложности. Мне казалось, что для построения минимального дерева на самом деле требовалось использовать какую-то нетривиальную комбинаторную оптимизацию, но я не видел, каким образом можно было бы включить упорядочивание правил слева направо в доказательство сложности. Более того, я не мог припомнить ни одной NP-полной задачи, содержащей такое ограничение в виде упорядочивания слева направо. В конце концов, если бы тот или иной набор из $n$ правил содержал позицию символа, общую для всех правил, то она должна была бы быть исследованной первой в любом дереве минимального размера. Так как правила были упорядоченными, то каждый узел в поддереве должен представлять корень серии последовательных правил. Таким образом, существовало только $\binom{n}{2}$ узлов, которые было возможно выбрать из этого дерева...

Есть! Это и было ключевым аспектом решения.

На следующий день я снова встретился с моим коллегой: «Я не могу доказать, что ваша задача является NP-полной. Но что Вы скажете по поводу эффективного алгоритма динамического программирования для построения наилучшего возможного нагруженного дерева?» — я с удовольствием наблюдал, как недовольное выражение его лица сменилось улыбкой, когда он осознал, о чем идет речь. Эффективный алгоритм для получения требуемого решения было значительно лучше, чем доказательство невозможности такого решения.

Мое рекуррентное соотношение работало следующим образом. Допустим, что у нас имеется $n$ упорядоченных голов правил $s_1$, ..., $s_n$, каждая из которых имеет $m$ аргументов. Выборка в $p$-й позиции, $1 \le p \le m$, разделяет головы правил на серии $R_1$, ..., $R_r$, где каждое правило в отдельной серии $R_x = s_1$, ..., $s_j$ имеет одинаковое значение символа $s_i[p]$. Правила в каждой серии должны быть последовательными, поэтому существует только $\binom{n}{2}$ возможных серий, о которых нужно заботиться. Стоимостью выборки в позиции $p$ является стоимость завершения обработки деревьев, формируемых каждой созданной серией, плюс одно ребро для каждого дерева, чтобы связать его с зондированием $p$:

$$C[i,j] = \min_{p=1}^{m}\left(\sum_{k=1}^{r}(C[i_k, j_k]+1)\right).$$

Один из моих аспирантов немедленно приступил к реализации этого алгоритма, чтобы можно было сравнить его работу с работой эвристического алгоритма заказчика. На многих входных экземплярах как оптимальный, так и «жадный» алгоритмы создавали одинаковое нагруженное дерево. Но для некоторых экземпляров производительность алгоритма динамического программирования была на 20% лучше, чем производительность «жадного» алгоритма, т. е. на 20% лучше, чем «очень хорошая работа на практике». Время компилирования алгоритма динамического программирования было несколько большим, чем «жадного» алгоритма, но при оптимизации компилирования всегда лучше обменять немного большее время компиляции на лучшее время исполнения программы. Стоит ли 20% улучшения производительности этих усилий? Это зависит от конкретной ситуации. Насколько полезной была бы для вас 20-процентная надбавка в вашей зарплате?

То обстоятельство, что правила должны были оставаться упорядоченными, оказалось решающим фактором, который мы использовали в решении с применением динамического программирования. Фактически при отсутствии этого обстоятельства я смог доказать, что задача действительно была NP-полной при произвольном упорядочении правил.

#### *Подведение итогов*

Глобальное оптимальное решение (полученное, например, с помощью динамического программирования) часто заметно лучше, чем решение, полученное посредством типичного эвристического алгоритма. Насколько важным является такое улучшение, зависит от конкретного приложения, но оно никогда не будет лишним.

## Замечания к главе

Методика динамического программирования впервые представлена Беллманом (Bellman) в [Bel58]. Алгоритм определения расстояния редактирования первоначально был представлен Вагнером и Фишером (Wagner, Fischer) в книге [WF74]. Более быстрый алгоритм для задачи разделения книг рассматривается в книге Ханна (Khanna) и прочих [KMS97].

Такие методики, как динамическое программирование и поиск с возвратом, можно использовать для создания эффективных в наихудшем случае (хотя все же не с полиномиальным временем исполнения) алгоритмов для решения многих NP-полных задач. Обзор этих методик см. в книге Дауни (Downey) и Феллоуза (Fellows) [DF12] и книге Уоджинджера (Woeginger) [Woe03].

Более подробная информация об историях из жизни предоставляется в опубликованных докладах. В частности, дополнительную информацию по задаче минимизации нагруженных деревьев в Prolog, которая рассматривалась в *разд. 10.10*, можно найти в докладе Доусона (Dawson) и прочих [DRR+95], а по алгоритму для равномерного распределения нагрузок по фазам (*разд. 10.6*) — в докладе Ванга (Wang) и прочих [WSR13]. Двумерные штрихкоды, рассматриваемые в *разд. 10.4*, были разработаны в значительной степени усилиями Тео Павлидиса (Theo Pavlidis) и Инджиуна Ванга (Ynjiun Wang) в Университете Stoney Brook. Подробности см. в [PSW92].

Алгоритм динамического программирования, рассматриваемый для решения задачи синтаксического анализа, называется CKYalgonthm, по именам его трех независимых разработчиков: Коки, Касами и Янгера (Cocke, Kasami, Younger). Подробности см. в [You67]. Обобщение синтаксического анализа для задачи вычисления расстояния редактирования рассматривается Ахо и Петерсоном (Aho, Peterson) в [AP72].

# 10.11. Упражнения

## Простые рекуррентные соотношения

1. *[3]* Ребенок бежит вверх по лестнице с $n$ ступеньками и может за раз подниматься на минимум 1 и максимум $k$ ступенек. Разработайте алгоритм для вычисления количества возможных способов, которыми ребенок может взбежать наверх, в зависимости от значений $n$ и $k$. Какова временная сложность вашего алгоритма?

2. *[3]* Представьте себе, что вы намереваетесь обворовать дома на улице из $n$ домов. Ценность добычи в доме $i$ составляет $m_i$, где $1 \leq i \leq n$, но соседние дома взламывать нельзя, поскольку взлом двух смежных домов вызовет подачу сигнала в полицейский участок. Разработайте эффективный алгоритм для максимизации вашей добычи без подачи сигнала тревоги полиции.

3. *[5]* Счет в баскетбольной игре ведется суммированием последовательности очков за успешные броски разных типов: 2 очка, 3 очка и 1 очко за штрафной бросок. Разработайте алгоритм для вычисления количества возможных комбинаций типов бросков по очкам (1, 2, 3), чтобы получить счет $n$. Например, счет $n = 5$ можно получить одной из следующих четырех комбинаций бросков: (5, 0, 0), (2, 0, 1), (1, 2, 0) и (0, 1, 1).

4. *[5]* Счет в баскетбольной игре ведется суммированием последовательности очков за успешные броски разных типов: 2 очка, 3 очка и 1 очко за штрафной бросок. Разработайте алгоритм для вычисления количества возможных последовательностей типов бросков по очкам, чтобы получить счет $n$. Например, для $n = 5$ существует 13 возможных последовательностей, включая 1-2-1-1, 3-2 и 1-1-1-1-1.

5. *[5]* Для матрицы размером $s \times t$, заполненной положительными числами, определите такой маршрут с левого верхнего узла в правый нижний узел, который минимизирует сумму всех чисел, содержащихся в узлах маршрута. За один ход разрешается перемещаться только на один узел вправо или вниз.

   a) Разработайте решение на основе алгоритма Дейкстры. Какова временная сложность такого решения в зависимости от значений $s$ и $t$?

   b) Разработайте решение на основе динамического программирования. Какова временная сложность такого решения в зависимости от значений $s$ и $t$?

## Расстояние редактирования

6. *[3]* При вводе текста часто допускаются ошибки перестановки соседних символов — например, когда вместо `Steve` вводится `Setve`. Согласно традиционному определению расстояния редактирования для исправления таких ошибок требуются две замены.

   Включите в нашу функцию расстояния редактирования операцию обмена, чтобы такие ошибки перестановки можно было бы исправить за одну операцию.

7. *[4]* Дано три строки символов: $X$, $Y$ и $Z$, где $|X| = n$, $|Y| = m$ и $|Z| = n + m$. Строка $Z$ называется *перетасовкой* (shuffle) строк $X$ и $Y$ тогда и только тогда, когда ее можно создать, перемешивая символы строк $X$ и $Y$ таким образом, чтобы в получившейся строке сохранялся первоначальный (слева направо) порядок исходных строк.

   a) Докажите, что строка `cchocohilaptes` является перетасовкой строк `chocolate` и `chips`, но строка `chocochilatspe` — нет.

   b) Предоставьте эффективный алгоритм динамического программирования для определения, является ли строка $Z$ перетасовкой строк $X$ и $Y$. (Подсказка: значения создаваемой вами матрицы динамического программирования должны быть булевыми, а не числовыми.)

8. *[4]* Самой длинной общей *подстрокой* (не подпоследовательностью) двух строк $X$ и $Y$ является самая длинная строка, которая входит в виде серии последовательных символов в обе строки. Например, самой длинной общей подстрокой строк `photograph` и `tomography` является строка `ograph`.

   a) Пусть $n = |X|$ и $m = |Y|$. Предоставьте алгоритм динамического программирования с временной сложностью $\Theta(nm)$ для поиска самой длинной общей подстроки, основанный на алгоритме поиска самой длинной общей подпоследовательности или вычисления расстояния редактирования.

   b) Предоставьте более простой алгоритм с временной сложностью $\Theta(nm)$, в котором не используется динамическое программирование.

9. *[6]* Самой *длинной общей подпоследовательностью* (longest common subsequence — LCS) двух последовательностей $T$ и $P$ называется такая самая длинная последовательность $L$, которая является подпоследовательностью как последовательности $T$, так и последовательности $P$. Кратчайшей общей надпоследовательностью (shortest common supersequence — SCS) последовательностей $T$ и $P$ называется такая кратчайшая последовательность $L$, подпоследовательностями которой являются как последовательность $T$, так и последовательность $P$.

   a) Предоставьте эффективный алгоритм поиска LCS и SCS для двух последовательностей.

   b) Пусть $d(T, P)$ будет минимальным расстоянием редактирования между строками $T$ и $P$ при условии, что замены запрещены, т. е. разрешены только вставка и удаление символов. Докажите, что $d(T, P) = |SCS(T, P)| - |LSC(T, P)|$, где $|SCS(T, P)|$ и $|LSC(T, P)|$ обозначают соответственно размер кратчайшей общей надпоследовательности и самой длинной общей подпоследовательности $T$ и $P$.

10. *[5]* Имеется $n$ фишек для игры в покер, упорядоченных в две стопки, в которых можно видеть ребра всех фишек. Фишки могут быть одного из трех цветов: красного (R, red), зеленого (G, green) или синего (B, blue). За один ход выбираем один из трех цветов и снимаем сверху обеих стопок все фишки этого цвета. Цель игры — убрать все фишки из обеих стопок за минимальное количество ходов.

    Рассмотрим, например, стопки (RRGG и GBBB). Убрать все фишки из этих стопок можно за три хода: красный, зеленый, синий. Разработайте алгоритм динамического программирования с временем исполнения $O(n^2)$ для вычисления наилучшей стратегии для решения задачи с такой парой стопок фишек.

## «Жадные» алгоритмы

11. *[4]* Пусть $P_1, P_2, \ldots P_n$ — это $n$ программ, которые нужно сохранить на диске емкостью $D$ Мбайт. Для хранения программы $P_i$ требуется $s_i$ Мбайт дискового пространства. Сохранить все программы на диске нельзя, т. к. $D < \sum_{i=1}^{n} s_i$.

    a) Можно ли максимизировать количество сохраняемых на диске программ с помощью «жадного» алгоритма, который выбирает программы в порядке неубывающего дискового пространства $s_i$? Предоставьте доказательство или контрпример.

    b) Можно ли сказать, что «жадный» алгоритм, выбирающий программы в порядке неубывающего дискового пространства $s_i$, использует наибольшее возможное дисковое пространство? Предоставьте доказательство или контрпример.

12. *[5]* В Соединенных Штатах используются монеты достоинством в 1, 5, 10, 25 и 50 центов. Теперь рассмотрим страну, в которой используются монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Нам нужен алгоритм, который дает сдачу размером в $n$ единиц, используя наименьшее количество монет этой страны.

    a) При решении этой задачи «жадный» алгоритм многократно выбирает монету самого большого достоинства, не превышающую текущий размер сдачи, до тех пор, пока размер сдачи не станет равен нулю. Докажите, что «жадный» алгоритм не всегда выбирает наименьшее количество монет для страны с достоинством монет в {1, 6, 10} единиц.

    b) Предоставьте эффективный алгоритм, который правильно определяет наименьшее количество монет, которое требуется, чтобы дать сдачу стоимостью в $n$ единиц, используя монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Выполните анализ времени исполнения этого алгоритма.

13. *[5]* В Соединенных Штатах используются монеты достоинством в 1, 5, 10, 25 и 50 центов. Теперь рассмотрим страну, в которой используются монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Нам нужно определить количество $C(n)$ разных способов дать сдачу стоимостью $n$ единиц. Например, для страны с монетами достоинством в {1, 6, 10} единиц имеем $C(5) = 1$, $C(6) = 2$, ..., $C(9) = 2$, $C(10) = 3$ и $C(12) = 4$.

    a) Сколько существует способов дать сдачу стоимостью в 20 единиц монетами достоинством в {1, 6, 10} единиц?

    b) Предоставьте эффективный алгоритм для вычисления $C(n)$ и выполните анализ его сложности. (Подсказка: подумайте о вычислении $C(n, d)$ — т. е. о вычислении количества способов дать сдачу стоимостью $n$ единиц, используя монеты наивысшего достоинства $d$. Будьте внимательны, чтобы не дать лишнюю сдачу.)

14. *[6]* Рассмотрим *задачу планирования загрузки однопроцессорной системы* с количеством $n$ заданий $J$. Для каждого задания $i$ установлено время обработки $t_i$ и крайнее время завершения $d_i$. Допустимым расписанием исполнения заданий является такая перестановка заданий, что при исполнении заданий в этом порядке каждое задание завершается до крайнего времени его завершения. «Жадный» алгоритм для решения задачи планировки загрузки однопроцессорной системы выбирает первым задание с самым ранним крайним временем завершения.

    Докажите, что если допустимое расписание существует, то создаваемое таким «жадным» алгоритмом расписание является допустимым.

## Числовые задачи

15. *[3]* Имеется пруток длиной $n$ дюймов и таблица цен, предлагаемых за отрезок прутка размером $n$ или меньше. Разработайте эффективный алгоритм для вычисления максимальной прибыли, которую можно получить, разрезав пруток на отрезки и продав их.

    Например, для прутка длиной $n = 8$ со следующими ценами на отрезки:

| Длина | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Цена | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 |

    максимальная прибыль, получаемая за два отрезка длиной 2 и 6, будет 22.

16. *[5]* Ваш начальник составил выражение с $n$ членами для вычисления вашего ежегодного бонуса и разрешил вам расставить в нем скобки любым способом. Разработайте эффективный алгоритм для расстановки скобок в выражении, чтобы максимизировать его значение.

    Например, в выражении

    $$6 + 2 \times 0 - 4$$

    скобки можно расставить несколькими способами, дающими значения от –32 до 2.

17. *[5]* Для заданного натурального числа $n$ разработайте эффективный алгоритм вычисления наименьшего количества полных квадратов (т. е. 1, 4, 9, 16, . . .), сумма которых составляет $n$. Какова временная сложность вашего алгоритма?

18. *[5]* Для заданного массива $A$ из $n$ целых чисел разработайте эффективный алгоритм вычисления суммы наибольшей непрерывной подпоследовательности элементов массива. Например, для массива A = [–3, 2, 7,–3, 4,–2, 0, 1] сумма наибольшей подпоследовательности будет 10 (для подпоследовательности элементов со второго по пятый).

19. *[5]* Между двумя автомобилями необходимо распределить $m$ чемоданов, при этом вес $i$-го чемодана составляет $w_i$. Разработайте эффективный алгоритм для распределения чемоданов, чтобы в каждом автомобиле их вес был одинаковым, если это возможно.

20. *[6] Задача о рюкзаке* задается таким образом: имея множество целых чисел $S = \{s_1, s_2, ..., s_n\}$ и целевое число $T$, найти такое подмножество множества $S$, сумма которого в точности равна $T$. Например, множество $S = \{1, 2, 5, 9, 10\}$ содержит подмножество, сумма членов которого составляет $T = 22$, но не $T = 23$. Предоставьте алгоритм динамического программирования для решения задачи о рюкзаке с временем исполнения $O(nT)$.

21. *[6]* В *задаче разделения множества целых чисел* требуется выяснить, содержит ли множество целых чисел $S = \{s_1, ..., s_n\}$ такое подмножество $I \subset S$, для которого

    $$\sum_{i \in I} s_i = \sum_{i \notin I} s_i.$$

    Пусть $\sum_{i \in S} s_i = M$. Предоставьте алгоритм динамического программирования с временем исполнения $O(nM)$ для решения задачи разделения множества целых чисел.

22. *[5]* Допустим, что имеется $n$ чисел (некоторые из них, возможно, отрицательные), расположенных по кругу. Разработайте эффективный алгоритм поиска наибольшей суммы смежных чисел в дуге.

23. *[5]* Некий язык для обработки строк позволяет разбивать строку на две части. Стоимость этого разбиения равна $n$ единицам времени, поскольку для этого требуется выполнить копирование старой строки. Программист хочет разбить строку на несколько частей, при этом общее время выполнения разбиения может зависеть от порядка, в котором осуществляется разбиение. Допустим, например, что мы хотим разбить 20-символьную строку в позициях после символов 3, 8 и 10. Если разбиения осуществляются в порядке слева направо, то первое разбиение стоит 20 единиц времени, второе — 17, а третье — 12, что дает общую стоимость в 49 единиц времени. Если же разбиения осуществляются в порядке справа налево, то первое разбиение стоит 20 единиц времени, второе — 10, а третье — 8, что дает общую стоимость в 38 единиц времени. Предоставьте алгоритм динамического программирования, который берет в качестве входа список позиций символов и определяет самое дешевое разбиение за время $O(n^3)$.

24. *[5]* Рассмотрим следующий способ сжатия данных. Имеется таблица, содержащая $m$ текстовых строк, каждая длиной самое большее $k$ символов. Требуется закодировать строку данных $D$ длиной $n$ символов, используя для этого наименьшее возможное количество текстовых строк из таблицы. Например, если таблица содержит строки (*a*, *ba*, *abab*, *b*), а строка данных имеет вид *bababbaababa*, то наилучшим способом кодирования будет (*b*, *abab*, *ba*, *abab*, *a*), в котором используются всего пять строк из таблицы. Предоставьте алгоритм с временем исполнения $O(nmk)$ для определения длины наилучшей кодировки. Можно полагать, что каждую кодируемую строку можно выразить, по крайней мере, одной комбинацией строк в таблице.

25. *[5]* Традиционный матч мирового чемпионата по шахматам состоит из 24 игр. Если матч заканчивается вничью, то текущий чемпион сохраняет свой титул. Каждая игра может быть выиграна, проиграна или сыграна вничью, где выигрыш равен 1, проигрыш — 0, а ничья — 1/2. Цвет фигур меняется каждую игру. Белые ходят первыми и поэтому имеют преимущество. В первой игре чемпион играет белыми. Его шансы выигрыша, ничьей и проигрыша равны $w_w$, $w_d$ и $w_l$ при игре белыми и $b_w$, $b_d$ и $b_l$ при игре черными соответственно.

    a) Напишите рекуррентное соотношение вероятности удержания чемпионом своего титула. Полагается, что в матче осталось сыграть $g$ игр и что чемпиону нужно набрать $i$ очков (количество которых может быть кратным 1/2).

    b) На основе этого рекуррентного соотношения разработайте алгоритм динамического программирования для вычисления вероятности сохранения чемпионом своего титула.

    c) Выполните анализ временной сложности вашего алгоритма для матча из $n$ игр.

26. *[8]* Если бросить яйцо с достаточной высоты, оно разобьется. В частности, в любом достаточно высоком здании должен быть $f$-й этаж, при падении с которого яйцо разобьется, но при падении с $(f-1)$-го этажа — нет. Если яйцо всегда разбивается, то тогда $f = 1$. Если яйцо никогда не разбивается, тогда $f = n + 1$.

    Нужно найти критический этаж $f$ в $n$-этажном здании. Для этого можно выполнять только одну операцию — бросать яйцо с определенного этажа и наблюдать за результатами. Надо выполнить как можно меньше таких операций, но в любом случае требуется потратить не больше чем $k$ яиц. Разбитые яйца снова использовать нельзя. Пусть $E(k, n)$ означает минимальное количество бросков, достаточное для получения решения.

    a) Докажите, что $E(1, n) = n$.

    b) Докажите, что $E(k, n) = \Theta(n^{1/k})$.

c) Выведите рекуррентное соотношение для $E(k, n)$. Каким будет время исполнения динамической программы для вычисления $E(k, n)$?

## Задачи на графы

27. *[4]* Рассмотрим город, улицы в котором задаются решеткой $X \times Y$. Нам нужно пройти из верхнего левого угла решетки в нижний правый угол. К сожалению, в городе имеются районы с плохой репутацией, и мы не хотим проходить по улицам в этих районах. Имеется матрица *bad* размером $X \times Y$, в которой $bad[i, j] = yes$ тогда и только тогда, когда перекресток улиц $i$ и $j$ находится в районе, через который мы не хотим проходить.

    a) Приведите пример такого содержимого матрицы *bad*, для которого не существует пути между требуемыми точками без прохождения через плохие районы.

    b) Предоставьте алгоритм сложностью $O(XY)$ для поиска пути, позволяющего избежать нежелательных районов.

    c) Предоставьте алгоритм сложностью $O(XY)$ для поиска *кратчайшего* пути, позволяющего избежать нежелательных районов. Полагается, что все кварталы одинаковой длины. Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $O(X^2Y^2)$.

28. *[5]* Даны такие же условия, что и в предыдущей задаче, — т. е. город, улицы в котором определены решеткой $X \times Y$. Нам нужно пройти из верхнего левого угла города/решетки к нижнему правому углу. Плохие районы определены матрицей *bad* размером $X \times Y$, в которой $bad[i, j] = yes$ тогда и только тогда, когда перекресток улиц $i$ и $j$ находится в районе, через который мы не хотим проходить.

    Если бы в городе не было неблагополучных районов, которые мы вынуждены избегать, то длина кратчайшего пути между указанными точками была бы равной $(X - 1) + (Y - 1)$ кварталам. Более того, было бы много таких путей, каждый из которых состоял бы только из движений вправо и вниз.

    Разработайте алгоритм, который принимает в качестве входа массив *bad* и возвращает *количество* безопасных путей длиной $X + Y - 2$. Чтобы решение было засчитано полностью, время исполнения алгоритма должно быть равным $O(XY)$.

29. *[5]* Требуется уложить в штабель $n$ ящиков, при этом $i$-й ящик имеет ширину $w_i$, высоту $h_i$ и длину $d_i$. Ящики укладываются в штабель с одинаковой ориентацией их габаритов и только таким образом, чтобы все размеры каждого следующего ящика были меньшими, чем все размеры предыдущего. Разработайте эффективный алгоритм для создания наивысшего возможного штабеля ящиков, высота которого является суммой высот всех ящиков в штабеле.

## Задачи по разработке

30. *[4]* В библиотеке нужно разместить на полках $n$ книг в установленном в каталоге порядке. Поэтому мы можем представить позицию конкретной книги как $b_i$, ее толщину как $t_i$, а высоту как $h_i$, где $1 \leq i \leq n$. Все полки в библиотеке имеют одинаковую длину $L$.

    Допустим, что все книги имеют одинаковую высоту $h$ (т. е. $h = h_i$ для всех $i$), а расстояние между полками больше чем $h$, вследствие чего любую книгу можно поставить на любую полку. «Жадный» алгоритм поставит на первую полку наибольшее возможное количество книг, пока не будет достигнуто такое наименьшее значение $i$, для которого

книга $b_i$ не будет вмещаться на эту полку, после чего повторит процедуру с последующими полками. Докажите, что «жадный» алгоритм всегда находит порядок размещения книг на минимальное количество полок, и укажите его временную сложность.

31. *[6]* Эта задача является вариантом предыдущей. Здесь книги разной высоты, но высоту каждой полки можно подогнать под высоту самой высокой книги на ней. При этом стоимость определенного размещения является суммой высот наивысших книг на каждой полке.

    - Предоставьте пример, показывающий, что «жадный» алгоритм, заполняющий каждую полку насколько возможно, не всегда дает минимальную общую высоту.
    - Разработайте алгоритм для решения этой задачи и выполните анализ его временной сложности. (Подсказка: используйте динамическое программирование.)

32. *[5]* Имеется линейная клавиатура со строчными буквами и цифрами. Первые левые 26 клавиш клавиатуры отведены буквам *a–z* в возрастающем порядке, следующие 10 — цифрам 0–9 в возрастающем порядке, следующие 30 — знакам препинания в заданном порядке, а последняя — пробелу. В начале печати любой последовательности символов указательный палец левой руки находится на клавише буквы *a*, а правой — на клавише пробела.

    Разработайте алгоритм динамического программирования для вычисления наиболее эффективного (в терминах минимального перемещения используемых пальцев) способа для набора заданного текста длиной $n$ символов. Например, для набора текста `ABABABAB...ABAB` оба пальца нужно переместить к левому краю клавиатуры. Какова временная сложность вашего алгоритма в зависимости от длины текста $n$ и количества клавиш $k$?

33. *[5]* Вы возвратились из будущего с массивом $G$, где элемент $G[i]$ содержит информацию о цене акций Google через $i$ дней после текущей даты и $1 \leq i \leq n$. Но эту информацию можно использовать для выполнения только одной транзакции (т. е. или покупки, или продажи одной акции) в день.

    Разработайте эффективный алгоритм для вычисления последовательности операций покупок и продаж, максимизирующей вашу прибыль. При этом можно продавать только акции, которыми вы владеете.

34. *[8]* Дана строка из $n$ символов $S = s_1 \ldots s_n$, которая представляет текстовый документ, из которого были удалены все пробелы. Например, `itwasthebestoftimes`.

    a) Необходимо восстановить документ, используя для этого словарь в виде булевой функции *dict*($w$), которая возвращает *истина* тогда и только тогда, когда $w$ является действительным словом. Предоставьте алгоритм с временем исполнения $O(n^2)$, чтобы определить, возможно ли восстановить строку $S$ в виде последовательности действительных слов. При этом предполагается, что вызовы функции *dict*($w$) занимают единицу времени.

    b) Теперь предположим, что у вас есть словарь в виде множества из $m$ слов, длина каждого из которых не больше $l$. Предоставьте эффективный алгоритм, чтобы определить, возможно ли восстановить строку $S$ в виде последовательности действительных слов. Укажите время исполнения этого алгоритма.

35. *[8]* Сыграйте в такую игру. Берется $n$-разрядное целое число $N$. В каждом ходе можно взять или первую, или последнюю цифру от оставшейся части $N$ и добавить ее к своему

счету, а затем ваш соперник делает то же самое с оставшейся частью числа. Ходы делаются до тех пор, пока в числе не останется цифр.

Разработайте эффективный алгоритм для вычисления наибольшего возможного счета, который может получить первый игрок для заданной цифровой последовательности $N$. При этом предполагается, что второй игрок всегда выполняет наилучшие возможные ходы.

36. *[6]* Имеется массив из $n$ действительных чисел, для которого нужно найти подмассив последовательных элементов с максимальной суммой. Например, в массиве

$$[31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$$

подмассивом последовательных элементов с максимальной суммой будут с третьего по седьмой элементы, а именно: 59 + 26 + (–53) + 58 + 97 = 187. Если все элементы массива положительные, то решением является весь массив, а когда все элементы отрицательные, то решением является пустой подмассив с суммой элементов, равной нулю.

- Разработайте простой и ясный алгоритм с временем исполнения $\Theta(n^2)$ для поиска подмассива последовательных элементов с максимальной суммой.
- Далее разработайте для решения этой же задачи алгоритм динамического программирования с временем исполнения $\Theta(n)$. Чтобы решение было зачтено частично, можно разработать алгоритм типа «разделяй и властвуй» с временем исполнения $O(n \log n)$.

37. *[7]* Даны алфавит из $k$ символов, строка $x = x_1 x_2 \ldots x_n$ из символов этого алфавита и таблица умножения символов алфавита:

| | *a* | *b* | *c* |
|---|---|---|---|
| *a* | *a* | *c* | *c* |
| *b* | *a* | *a* | *b* |
| *c* | *c* | *c* | *c* |

Нужно определить, возможно ли заключить части строки в скобки таким образом, чтобы в результате получить $a$, где $a$ является символом алфавита. Таблица умножения не обладает ни перестановочным, ни ассоциативным свойствами, вследствие чего порядок выполнения операций умножения имеет значение.

Для примера рассмотрим приведенную таблицу умножения и строку *bbbba*. Заключение частей строки в скобки в виде $(b(bb))(ba)$ дает результат $a$, но в виде $((((bb)b)b)a)$ дает результат $c$.

Предоставьте алгоритм с полиномиальной временной сложностью по отношению к $n$ и $k$ для определения, можно ли заключить в скобки части заданной строки, чтобы получить целевой символ согласно приведенной таблице умножения.

38. *[6]* Даны константы $\alpha$ и $\beta$. Допустим, что стоимость движения влево в двоичном дереве поиска равна $\alpha$, а вправо — $\beta$. Разработайте алгоритм для создания дерева с оптимальной ожидаемой стоимостью запроса для ключей $k_1, \ldots, k_n$, вероятность просмотра каждого из которых составляет $p_1, \ldots, p_n$.

## Задачи, предлагаемые на собеседовании

39. *[5]* Для определенного набора достоинств монет найти наименьшее количество монет, которым можно дать сдачу определенного размера.
40. *[5]* Имеется массив из $n$ чисел, каждое из которых может быть положительным, отрицательным или нулем. Предоставьте эффективный алгоритм для определения индексов элементов $i$ и $j$ для получения максимальной суммы от $i$-го до $j$-го числа.
41. *[7]* При вырезании символа из страницы журнала также вырезается символ на обратной стороне страницы. Предоставьте алгоритм для определения, возможно ли создать определенную строку из вырезанных из этого журнала символов. Можно полагать, что имеется функция, которая позволяет определить символ на обратной стороне страницы и его позицию для любого символа на лицевой стороне страницы.

## LeetCode

1. **https://leetcode.com/problems/binary-tree-cameras/**
2. **https://leetcode.com/problems/edit-distance/**
3. **https://leetcode.com/problems/maximum-product-of-splitted-binary-tree/**

## HackerRank

1. **https://www.hackerrank.com/challenges/ctci-recursive-staircase/**
2. **https://www.hackerrank.com/challenges/coin-change/**
3. **https://www.hackerrank.com/challenges/longest-increasing-subsequent/**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org**:

1. «Is Bigger Smarter?», глава 11, задача 10131.
2. «Weights and Measures», глава 11, задача 10154.
3. «Unidirectional TSP», глава 11, задача 10116.
4. «Cutting Sticks», глава 11, задача 10003.
5. «Ferry Loading», глава 11, задача 10261.

# ГЛАВА 11

# NP-полнота

В этой главе рассматриваются методы доказательства того, что для решения такой задачи не может существовать эффективного алгоритма. Читателям, не склонным к теоретизированию, необходимость доказывать что-либо, будет, скорее всего, неприятна, а сама мысль тратить время на доказательство чего-то несуществующего покажется абсолютно неприемлемой. Однако, если мы не знаем, как решить какую-то задачу, бывает небесполезно выяснить, что она не имеет решения.

Теория NP-полноты является чрезвычайно полезным инструментом разработчика алгоритмов, даже когда дает отрицательные результаты. В частности, теория NP-полноты позволяет разработчику алгоритмов тратить усилия более эффективно, показав, что поиск эффективного алгоритма для той или иной задачи обречен на неудачу. Зато, если вы потерпели неудачу, пытаясь доказать нерешаемость задачи, вы можете ожидать, что существует эффективный алгоритм для ее решения. Две истории из жизни, приведенные в *главе 10*, описывают случаи, закончившиеся благополучно, несмотря на неоправданные жалобы разработчиков на сложность задачи.

Теория NP-полноты также позволяет распознать свойства, которые делают определенную задачу сложной. Это может дать нам представление о том, как следует моделировать ее разными способами или использовать ее более благоприятные характеристики. Понимание того, какие задачи являются сложными, исключительно важно для разработчиков алгоритмов, и оно может быть приобретено только в попытках доказать сложность задач.

Для доказательства сложности задач мы воспользуемся таким фундаментальным принципом, как *сведение* (reduction), показывая, что две задачи в действительности одинаковые. Для демонстрации этой идеи мы рассмотрим последовательность сведений, каждое из которых дает или эффективный алгоритм для решения одной задачи, или доказательство, что такого алгоритма не может существовать для решения другой. В этой главе также содержится краткое введение в теоретико-сложностные аспекты NP-полноты — одного из наиболее фундаментальных понятий теории вычислительных систем.

## 11.1. Сведение задач

В этой книге нам встретилось несколько задач, для решения которых мы не могли найти никаких эффективных алгоритмов. Теория NP-полноты предоставляет инструменты, с помощью которых можно показать, что на определенном уровне все эти задачи в действительности являются одной и той же задачей.

Ключевым принципом для демонстрации сложности задачи является *сведение*, или преобразование одной задачи в другую. С объяснением этой идеи может помочь такая

аллегория NP-полноты. Несколько парней по очереди дерутся друг с другом, чтобы выяснить, кто из них «круче». Адам победил Билла, который затем одолел Дуэйна. Кто же из этих троих самый «крутой», и есть ли смысл вообще говорить об этом? Без какого-либо внешнего стандарта ответить невозможно. Если эти выяснения происходят во дворе детского сада, тогда их результаты не имеют большого значения. Но вот если бы оказалось что Дуэйн это Дуэйн Дуглас Джонсон по прозвищу Скала[1], в «крутости» которого никто не сомневается, то Адам и Билл были бы не менее «крутыми», чем он. В этой аллегории NP-полноты каждая драка представляет сведение, а Дуэйн Дуглас Джонсон играет роль задачи *выполнимости* (satisfiability), т. е. достоверно трудно решаемой задачи.

Итак, сведение — это алгоритм преобразования одной задачи в другую. Чтобы описать сведение, нам нужно использовать достаточно строгие определения. Алгоритмическая *задача* представляет собой вопрос общего характера, для которого даются входные параметры и условия, формулирующие, что можно считать удовлетворительным ответом или решением. *Экземпляром* (instance) называется задача, для которой указаны параметры входа. Разницу между задачей и ее экземпляром можно продемонстрировать на следующем примере:

**Задача.** Задача коммивояжера.

**Вход.** Взвешенный граф $G$.

**Выход.** Какой маршрут ($v_1$, $v_2$, ..., $v_n$) минимизирует $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

Любой взвешенный граф определяет экземпляр задачи коммивояжера, а каждый конкретный *экземпляр* имеет по крайней мере один маршрут с минимальной стоимостью. Для общей *задачи* коммивояжера требуется найти алгоритм определения оптимального маршрута для любого возможного экземпляра задачи.

## 11.1.1. Ключевая идея

Теперь рассмотрим две алгоритмические задачи, носящие названия *Bandersnatch* и *Bobilly*. Допустим, что у нас имеется алгоритм решения задачи Bandersnatch (листинг 11.1).

**Листинг 11.1. Решение задачи Bandersnatch**

```
Bandersnatch(G)
  Преобразуем G в экземпляр Y задачи Bo-billy
  Вызываем процедуру Bo-billy для решения экземпляра Y
  Возвращаем результат Bo-billy(Y) в качестве ответа для Bandersnatch(G)
```

Этот алгоритм выдаст *правильное* решение задачи Bandersnatch при условии, что при ее преобразовании в задачу Bo-billy всегда сохраняется правильность ответа. Иными словами, при условии, что преобразование имеет свойство, что для любого экземпляра $G$:

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y).$$

[1] См. **https://ru.wikipedia.org/wiki/Джонсон,_Дуэйн**. — *Прим. пер.*

Преобразование экземпляров одного типа задачи в экземпляры задачи другого типа с сохранением правильных ответов и называется *сведением* (reduction).

Теперь допустим, что при таком сведении экземпляр $G$ преобразуется в $Y$ за время $O(P(n))$. Тогда возможны два варианта:

- *если* процедура Bo-billy выполняется за время $O(P'(n))$, это дает алгоритм для решения задачи Bandersnatch за время $O(P(n) + P'(n))$, преобразовывающий задачу, а затем исполняющий процедуру Bo-billy для ее решения;
- *если* известно, что выражение $\Omega(P'(n))$ является нижним пределом при вычислении Bandersnatch (т. е. что не существует алгоритма для более быстрого решения этой задачи), тогда выражение $\Omega(P'(n) - P(n))$ *должно* быть нижним пределом для вычисления Bo-billy. Почему это так? Если *можно было бы* решить Bo-billy быстрее, то ранее описанное сведение нарушило бы нижний предел для решения Bandersnatch. Поскольку это невозможно, то не может существовать способа решения Bo-billy быстрее, чем заявлено.

По существу, такое сведение показывает, что задача Bo-billy не легче, чем задача Bandersnatch. Поэтому, если задача Bandersnatch сложная, то это означает, что задача Bo-billy также должна быть сложной. Для иллюстрации этого момента мы рассмотрим в этой главе несколько примеров сведения задач.

#### Подведение итогов

Посредством сведения можно показать, что две задачи являются, по сути, одинаковыми. Наличие эффективного алгоритма для решения одной задачи (или отсутствие такого) подразумевает наличие (или отсутствие) эффективного алгоритма для решения другой.

## 11.1.2. Задачи разрешимости

При сведении задача одного типа преобразуется в задачу другого типа таким образом, что ответы для всех экземпляров задачи являются идентичными. Задачи отличаются друг от друга *диапазоном* или *типом* возможных ответов. Решение задачи коммивояжера состоит из перестановки вершин, в то время как решения других задач могут возвращаться в виде строк или чисел, диапазон которых может быть ограничен положительными или целыми числами.

Диапазон решений наиболее интересного класса задач ограничен значениями *истина* или *ложь*. Эти задачи называются *задачами разрешимости* (decision problems). Удобно сводить одну задачу разрешимости к другой, поскольку допустимыми ответами как для начальной, так и для конечной задачи являются только *истина* или *ложь*.

К счастью, большинство представляющих интерес задач оптимизации можно сформулировать в виде задачи разрешимости, которая отражает суть вычислений. Например, задача разрешимости для задачи коммивояжера определяется таким образом:

**Задача.** Разрешимость задачи коммивояжера.

**Вход.** Взвешенный граф $G$ и целое число $k$.

**Выход.** Ответ, существует ли маршрут стоимостью $\leq k$.

Формулировка в виде задачи разрешимости отражает саму суть задачи коммивояжера в том смысле, что если существует эффективный алгоритм для решения задачи разре-

шимости, то с его помощью можно выполнить двоичный поиск для разных значений $k$ и быстро определить стоимость оптимального решения. А приложив некоторые усилия, по эффективному решению задачи разрешимости можно восстановить и саму перестановку узлов маршрута.

В дальнейшем мы будем, как правило, говорить о задачах разрешимости, потому что с ними легче работать, но при этом, они тем не менее отображают всю мощь теории NP-полноты..

## 11.2. Сведение для создания новых алгоритмов

Сведение одной задачи к другой — достойный способ создания новых алгоритмов из старых. Всегда, когда входные данные для задачи, *которую мы хотим решить*, можно преобразовать во входные данные для задачи, *которую мы знаем, как решить*, то процедура преобразования и известное решение образуют алгоритм для решения нашей задачи.

В этом разделе мы рассмотрим несколько примеров сведения, которые позволяют воспользоваться эффективными алгоритмами. Чтобы решить задачу $A$, мы выполняем сведение экземпляра задачи $A$ к экземпляру задачи $B$, после чего решаем этот экземпляр, используя эффективный алгоритм для решения задачи $B$. Общее время исполнения в таком случае равно времени, необходимому для выполнения сведения, плюс время для решения экземпляра задачи $B$.

### 11.2.1. Поиск ближайшей пары

В задаче *поиска ближайшей пары* требуется найти в множестве чисел пару чисел с наименьшей разницей между ними. Например, в множестве $S = \{10, 4, 8, 3, 12\}$ ближайшей парой будет подмножество (3, 4). Эту задачу можно преобразовать в задачу разрешимости, задав вопрос, является ли такая разница меньшей, чем некое пороговое значение. А именно:

**Вход.** Множество $S$ из $n$ чисел и пороговое значение $t$.

**Выход.** Решение, существует ли такая пара чисел $s_i, s_j \in S$, для которой $|s_i - s_j| \leq t$.

Задача поиска ближайшей пары решается применением простой сортировки, поскольку после сортировки члены ближайшей пары должны находиться рядом друг с другом. Отсюда следует такой алгоритм:

```
CloseEnoughPair(S, t)
   Sort S
   Is min_{1 ≤ i < n} |s_{i+1} - s_i| ≤ t?
```

По поводу этого простого сведения можно сделать несколько замечаний.

1. Сведенная к задаче разрешимости версия задачи сохраняет суть общей задачи, а это означает, что решить ее не легче, чем саму задачу поиска ближайшей пары.
2. Временная сложность алгоритма решения зависит от временной сложности сортировки. Например, если применить для сортировки алгоритм с временной сложностью $O(n \log n)$, то время поиска ближайшей пары будет равно $O(n \log n + n)$.

3. Это сведение и факт существования нижнего предела времени исполнения сортировки, равного $\Omega(n \log n)$, *не* доказывают, что в наихудшем случае достаточно близкую пару можно найти за время $\Omega(n \log n)$. Возможно, существует более быстрый подход — иной, нежели сортировка?
4. С другой стороны, если бы мы знали, что в наихудшем случае достаточно близкую пару можно найти за время $\Omega(n \log n)$, то это сведение было бы достаточным доказательством того, что сортировку нельзя осуществить быстрее, чем за время $\Omega(n \log n)$, поскольку такая более быстрая сортировка подразумевала бы более быстрый алгоритм поиска достаточно близкой пары.

## 11.2.2. Максимальная возрастающая подпоследовательность

В *главе 10* было продемонстрировано использование динамического программирования для решения разных задач, включая вычисление расстояния редактирования строки. Приведу краткий обзор этих задач.

**Задача.** Вычислить расстояние редактирования.

**Вход.** Последовательности целых чисел или символов $S$ и $T$. Стоимость каждой операции вставки обозначена как $c_{ins}$, удаления — $c_{del}$ и замены — $c_{sub}$.

**Выход.** Стоимость наиболее дешевой последовательности операций для преобразования последовательности $S$ в последовательность $T$.

Было показано, что расстояние редактирования можно использовать для решения многих других задач. Но эти алгоритмы часто можно воспринимать как алгоритмы сведения. Рассмотрим следующую задачу:

**Задача.** Найти максимальную возрастающую подпоследовательность.

**Вход.** Последовательность $S$ целых чисел или текстовых символов.

**Выход.** Такая максимальная последовательность позиций целых чисел $p_1, ..., p_m$, для которой $p_i < p_{i+1}$ и $S_{p_i} < S_{p_{i-1}}$.

В *разд. 10.3* было показано, что задачу поиска максимальной возрастающей подпоследовательности можно решить в виде частного случая задачи вычисления расстояния редактирования (листинг 11.2).

**Листинг 11.2. Поиск максимальной возрастающей подпоследовательности**

```
Longest Increasing Subsequence(S)
   T = Sort(S)
   c_ins = c_del = 1
   c_sub = ∞
   Return (|S| − EditDistance(S, T, c_ins, c_del, c_sub)/2)
```

Почему этот алгоритм дает правильный результат? Создавая вторую последовательность $T$ из элементов последовательности $S$, отсортированных в возрастающем порядке, мы обеспечиваем, что любая общая подпоследовательность должна быть возрас-

тающей. Если же замены не разрешены ни при каких обстоятельствах (поскольку $c_{sub} = \infty$), то в результате оптимального выравнивания последовательностей $S$ и $T$ мы находим их максимальную общую подпоследовательность и удаляем все прочее. Например, для преобразования последовательности $S = cab$ в последовательность $abc$ требуются две операции: удаление и вставка символа $c$, не попавшего в максимальную общую подпоследовательность. Длиной максимальной возрастающей подпоследовательности будет длина последовательности $S$ за вычетом половины стоимости этого преобразования.

Каковы последствия этого сведения? Исполнение самого сведения занимает время $O(n \log n)$ вследствие стоимости сортировки. Поскольку вычисление расстояния редактирования занимает время $O(|S| \cdot |T|)$, то временная сложность алгоритма поиска максимальной возрастающей подпоследовательности в $S$ будет квадратичной. В действительности, для поиска максимальной возрастающей подпоследовательности существует более быстрый алгоритм с временной сложностью $O(n \log n)$, основанный на удачно подобранных структурах данных, в то время как алгоритм вычисления расстояния редактирования в худшем случае имеет квадратичную временную сложность. Таким образом, в результате сведения задач мы получим простой, но неоптимальный алгоритм с полиномиальной временной сложностью.

## 11.2.3. Наименьшее общее кратное

При работе с целыми числами часто требуется найти *наименьшее общее кратное* и *наибольший общий делитель*. Говорят, что *a кратно b* (записывается: $b \mid a$), если существует такое целое число $d$, для которого $a = bd$. Тогда постановка этих двух задач выглядит так:

**Задача**. Найти наименьшее общее кратное (НОК).

**Вход.** Два натуральных числа: $x$ и $y$.

**Выход.** Наименьшее натуральное число $m$, которое кратно и числу $x$, и числу $y$.

Итак:

**Задача.** Найти наибольший общий делитель (НОД).

**Вход.** Два натуральных числа: $x$ и $y$.

**Выход.** Наибольшее натуральное число $d$, на которое делится как число $x$, так и число $y$.

Например, НОК(24, 36) = 72, а НОД(24, 36) = 12. Обе задачи можно с легкостью решить, разложив числа $x$ и $y$ на простые множители, но до сих пор не предложен эффективный алгоритм для разложения целых чисел на множители (см. *разд. 16.8*). К счастью, задачу поиска наибольшего общего делителя можно эффективно решить с помощью алгоритма Евклида, не прибегая при этом к разложению на множители. Этот рекурсивный алгоритм основан на двух наблюдениях. Первое:

если $(b \mid a)$, тогда НОД$(a, b) = b$.

Это должно быть вполне очевидным, поскольку если $a$ кратно $b$, тогда $a = bk$ для некоторого целого числа $k$, откуда следует, что НОД$(bk, b) = b$.

*И* второе:

если $a = bt + r$ для целых чисел $t$ и $r$, тогда НОД($a$, $b$) = НОД($b$, $r$).

Тогда для $a \geq b$ евклидовым алгоритмом многократно заменяем ($a$, $b$) на ($b$, $a$ mod $b$) до тех пор, пока не получим $b = 0$. Время исполнения этого алгоритм в наихудшем случае составляет $O(\log b)$.

Поскольку $x \cdot y$ является кратным как $x$, так и $y$, то НОК($x$, $y$) $\leq xy$. Существование меньшего общего кратного возможно лишь в том случае, если существует какое-либо нетривиальное общее кратное для $x$ и $y$. Это свойство в совокупности с алгоритмом Евклида обеспечивает эффективный способ для вычисления наименьшего общего кратного, как показано в следующем псевдокоде:

```
LeastCommonMultiple(x, y)
   Return (xy/НОД(x,y)).
```

Это сведение позволяет нам применить алгоритм Евклида для решения другой задачи.

## 11.2.4. Выпуклая оболочка (*)

В нашем следующем примере сведения «легкой» задачи (т. е. задачи, которая решается за полиномиальное время) мы используем сортировку для поиска выпуклой оболочки множества точек. Многоугольник называется *выпуклым*, если отрезок прямой линии, проведенной между любыми двумя точками внутри многоугольника $P$, находится полностью внутри многоугольника. Это возможно, когда многоугольник $P$ не содержит зазубрин или вогнутостей, поэтому выпуклые многоугольники имеют аккуратную форму. Выпуклая оболочка предоставляет удобный способ структурирования множества точек. Соответствующие приложения рассматриваются в *разд. 20.2*.

**Задача.** Найти выпуклую оболочку.

**Вход.** Множество $S$, состоящее из $n$ точек, лежащих в плоскости.

**Выход.** Наименьший выпуклый многоугольник, содержащий все точки множества $S$.

Здесь мы рассмотрим, как преобразовать экземпляр задачи сортировки (скажем, множества {13, 5, 11, 17}) в экземпляр задачи выпуклой оболочки. Это означает, что каждое число преобразуется в точку на плоскости, для чего значение $x$ отображается на точку $(x, x^2)$, — т. е. каждое целое число отображается на точку параболы $y = x^2$ (рис. 11.1).

А т. к. область сверху этой параболы является выпуклой, то каждая точка должна находиться на выпуклой оболочке. Кроме этого, поскольку соседние точки на выпуклой оболочке имеют соседние значения $x$, то выпуклая оболочка возвращает точки, отсортированные по $x$-координате, — т. е. первоначальные числа. Временная сложность алгоритма для создания и считывания точек, приведенного в листинге 11.3, равна $O(n)$.

**Листинг 11.3. Алгоритм создания и считывания точек выпуклой оболочки**

```
Sort(S)
Для каждого числа i∈S создаем точку (i, i²).
   Вызываем процедуру выпуклой оболочки для этого набора точек.
   Начиная с самой левой точки оболочки, считываем точки слева направо.
```

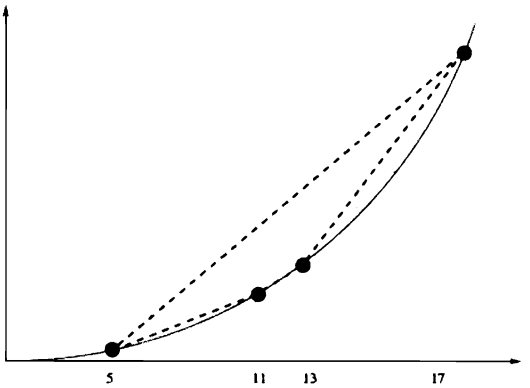



**Рис. 11.1.** Сведение задачи сортировки к задаче поиска выпуклой оболочки посредством отображения точек на параболе

Что все это означает? Вспомним, что нижний асимптотический предел сортировки равен $\Omega(n \lg n)$. Если бы временная сложность вычисления выпуклой оболочки могла быть лучшей, чем $n \lg n$, то это сведение подразумевало бы временную сложность сортировки лучшую, чем $\Omega(n \lg n)$, что нарушает наш нижний предел. Соответственно временная сложность вычисления выпуклой оболочки также должна быть равной $\Omega(n \lg n)$! Обратите внимание, что любой алгоритм для вычисления выпуклой оболочки, имеющий временную сложность $O(n \lg n)$, если его объединить с этим сведением, также предоставляет нам сложный, но правильный алгоритм сортировки с временем исполнения $O(n \lg n)$.

# 11.3. Простые примеры сведения сложных задач

Сведения, приведенные в *разд. 11.2*, демонстрируют преобразования между парой задач, для решения которых существуют эффективные алгоритмы. Но нас в основном интересует использование механизма сведения для доказательства сложности задачи, показывая, что сложность задачи Bo-billy как минимум такая же, как и сложность задачи Bandersnatch.

На этом этапе вам нужно просто поверить мне на слово, что задачи поиска *гамильтонова цикла* и *вершинного покрытия* являются сложными. Вся картина сведения (рис. 11.2) станет ясной по завершении изучения этой главы.

## 11.3.1. Гамильтонов цикл

Задача поиска гамильтонова цикла является одной из наиболее знаменитых задач в теории графов. В ней требуется найти маршрут, который проходит через каждую вершину графа только один раз. Эта задача имеет долгую историю и множество применений, что рассматривается в *разд. 19.5*. Формальное определение задачи следующее:




**Рис. 11.2.** Часть дерева сводимости для NP-полных задач. Сплошные линии обозначают сводимости, рассматриваемые в этой главе

**Задача**. Найти гамильтонов цикл.

**Вход.** Невзвешенный граф $G$.

**Выход.** Существует ли простой маршрут, который проходит через каждую вершину графа $G$ ровно один раз?

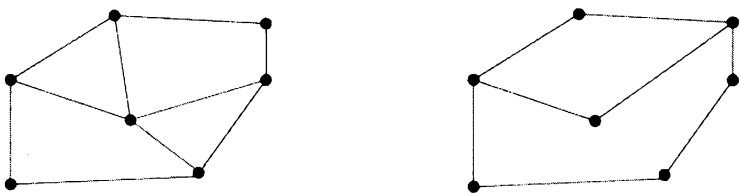
Задача поиска гамильтонова цикла имеет очевидное сходство с задачей коммивояжера. И в той и в другой требуется найти маршрут, который проходит через каждую вершину ровно один раз. Но между этими двумя задачами имеются и различия. Задача коммивояжера решается для взвешенных графов, а гамильтонова задача — для невзвешенных. Из показанного в листинге 11.4 сведения задачи гамильтонова цикла к задаче коммивояжера можно видеть, что сходств между этими задачами больше, чем различий.

**Листинг 11.4. Алгоритм сведения задачи поиска гамильтонова цикла к задаче коммивояжера**

```
HamiltonianCycle(G = (V,E))
  Создаем полный взвешенный граф G' = (V', E'), где V' = V.
  n=|V|
  for i = 1 to n do
    for j = 1 to n do
      if (i, j) ∈ E then w(i, j) = 1 else w(i, j) = 2
  Решаем задачу Traveling-Salesman-Decision-Problem(G', n)
```

Само сведение не несет в себе ничего сложного, причем преобразование из невзвешенного во взвешенный граф осуществляется таким образом, чтобы была обеспечена идентичность ответов для обеих задач.

Если граф $G$ содержит гамильтонов цикл $(v_1, ..., v_n)$, тогда этот же маршрут будет соответствовать $n$ ребрам в наборе ребер $E'$, вес каждого из которых равен единице. Это определяет маршрут для задачи коммивояжера в графе $G'$ весом ровно в $n$ единиц. Если граф $G$ не содержит гамильтонов цикл, тогда каждый маршрут в графе $G'$ должен содержать как минимум одно ребро весом в 2 единицы, что исключает наличие маршрута весом в $n$ единиц. На рис. 11.3, *а* представлен пример графа, содержащего гамильтонов цикл, а на рис. 11.3, *б* — графа, не содержащего гамильтонов цикл.



*а* *б*

**Рис. 11.3.** Примеры графов: *а* — граф содержит гамильтонов цикл; *б* — не содержит его

Такое сведение является эффективным, сохраняет истинность и имеет время исполнения $\Theta(n^2)$. Наличие эффективного алгоритма для решения задачи коммивояжера подразумевало бы наличие эффективного алгоритма для решения задачи поиска гамильтонова цикла, в то время как доказательство сложности задачи поиска гамильтонова цикла также подразумевало бы сложность задачи коммивояжера. Поскольку в нашем случае имеет место второй вариант, это сведение показывает, что задача коммивояжера является сложной, — по меньшей мере в той же степени, что и задача поиска гамильтонова цикла.

## 11.3.2. Независимое множество и вершинное покрытие

Задача о вершинном покрытии, которая рассматривается более подробно в *разд. 19.3*, заключается в поиске минимального набора вершин, с которым соприкасается хотя бы один конец каждого ребра графа. Формальное определение задачи следующее:

**Задача**. Найти вершинное покрытие.

**Вход**. Граф $G = (V, E)$ и целое число $k \leq |V|$.

**Выход**. Существует ли такое подмножество $S$, содержащее самое большее $k$ вершин, для которого каждое ребро $e \in E$ содержит по крайней мере одну вершину в множестве $S$?

Поиск «обычного» вершинного покрытия графа, т. е. покрытия, содержащего *все* вершины, представляет собой тривиальную задачу. Более сложной является задача охвата всех ребер с использованием минимально возможного набора вершин. Для графа, показанного на рис. ЦВ-11.4, вершинное покрытие образовано четырьмя из восьми его вершин.

Множество вершин $S$ графа $G$ является *независимым*, если не существует ребер $(x, y)$, для которых как вершина $x$, так и вершина $y$ являются элементами множества вершин $S$. Это означает, что ни одна из любой пары вершин независимого множества не соединена ребром. Поиск *какого-либо* независимого множества вершин также является тривиальной задачей — просто берем любую одну вершину. Как будет показано в *разд. 19.2*, независимое множество возникает при решении задач размещения точек обслуживания.

Постановка такой задачи выглядит таким образом:

**Задача**. Найти независимое множество.

**Вход**. Граф $G$ и целое число $k \leq |V|$.

**Выход**. Существует ли в графе $G$ множество из $k$ независимых вершин?

Как в задаче о вершинном покрытии, так и в задаче о независимом множестве нужно найти особое подмножество вершин, содержащее хотя бы один конец каждого ребра в первом случае и не содержащее концы ребер во втором. Если множество $S$ является вершинным покрытием графа $G$, тогда остальные вершины $V - S$ должны составлять независимое множество, поскольку если бы существовало ребро $(x, y)$, обе вершины которого находились бы в подмножестве $V - S$, то тогда множество $S$ не могло бы быть вершинным покрытием. Это позволяет нам выполнить сведение между этими двумя задачами (листинг 11.5).

**Листинг 11.5. Алгоритм для сведения задачи о независимом множестве и задачи о вершинном покрытии**

```
VertexCover(G, k)
    G' = G
    k'= |V| - k
    Решаем задачу IndependentSet(G', k ')
```

Опять же, простое сведение показывает, что сложность одной задачи как минимум такая же, как и сложность другой. Обратите внимание на то, что это преобразование осуществляется без какой бы то ни было информации об ответе. Преобразованию подвергается *вход*, а не решение. Это сведение демонстрирует, что из сложности задачи о вершинном покрытии вытекает сложность задачи о независимом множестве. В нашем конкретном случае преобразуемые задачи можно с легкостью поменять местами, и это доказывает, что обе задачи одинаково сложны.

**ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ:**

## Сложность общей задачи календарного планирования

**ЗАДАЧА**. Вспомните задачу календарного планирования съемок в фильмах (см. *разд. 1.2*). Там каждому фильму соответствовал один временной интервал, во время которого осуществлялись съемки фильма. Требовалось найти наибольший возможный набор неконфликтующих фильмов — т. е. фильмов, периоды съемок которых не пересекаются.

Общий вариант календарного планирования позволяет разбить выполнение одного задания на несколько временных интервалов. Например, задание $A$ (период исполнения

январь-март и май-июнь, не конфликтует с заданием *B* (период исполнения апрель-август), но *конфликтует* с заданием *C* (период исполнения июнь-июль).

Докажите, что *общая* задача календарного планирования является NP-полной, используя сведение от независимого множества.

**Задача**. Общая задача календарного планирования.

**Вход**. Множество $\{I_1, ..., I_n\}$, состоящее из *n множеств* линейных интервалов, — целое число *k*.

**Выход**. Можно ли из множества *I* выбрать подмножество из, по крайней мере, *k* взаимно непересекающихся *множеств* линейных интервалов?

**РЕШЕНИЕ**. Чтобы доказать сложность задачи, сначала нужно установить, какая задача играет роль Bandersnatch, а какая — роль Bo-billy. Здесь нам необходимо показать, как преобразовать *все* экземпляры задачи о независимом множестве в экземпляры общей задачи календарного планирования, т. е. в наборы непересекающихся линейных интервалов. Поэтому задача независимого множества будет задачей Bandersnatch, а общая задача календарного планирования — задачей Bo-billy.

Какое соответствие имеется между этими двумя задачами? В обоих случаях требуется выбрать наибольшее возможное подмножество: вершин в задаче о независимом множестве и проектов в задаче календарного планирования. Это обстоятельство наводит на мысль, что нужно преобразовать вершины в проекты. Более того, в обеих задачах требуется, чтобы выбранные элементы не конфликтовали, т. е. вершины не имели общее ребро, а временные интервалы проектов не пересекались. Соответствующее сведение показано в листинге 11.6.

**Листинг 11.6. Алгоритм для сведения задачи о независимом множестве к задаче календарного планирования**

```
IndependentSet(G, k)
    I = ∅
    Для i-го ребра (x, y), 1 ≤ i ≤ m
        Добавляем интервал [i, i + 0.5] к множеству интервалов I_x в I проекта  X
        Добавляем интервал [i, i + 0.5] к множеству интервалов I_y в I проекта  Y
    Возвращаем ответ к задаче GeneralMovieScheduling(I, k)
```

Мое решение строится таким образом. Для каждого из *m* ребер графа на линии создается интервал. Соответствующий каждой вершине проект будет содержать интервалы для смежных с ним ребер, как показано на рис. 11.5.

Каждая пара вершин, имеющая общее ребро (что не допускается в независимом множестве), определяет пару проектов, имеющих общий временной интервал (что не допускается в расписании съемок актера). Таким образом, наибольшие подмножества, удовлетворяющие условиям обеих задач, являются одинаковыми, а эффективный алгоритм для решения общей задачи календарного планирования дает нам быстрый алгоритм для решения задачи о независимом множестве. Соответственно, общая задача календарного планирования должна быть как минимум такой же сложной, как и задача о независимом множестве. ■

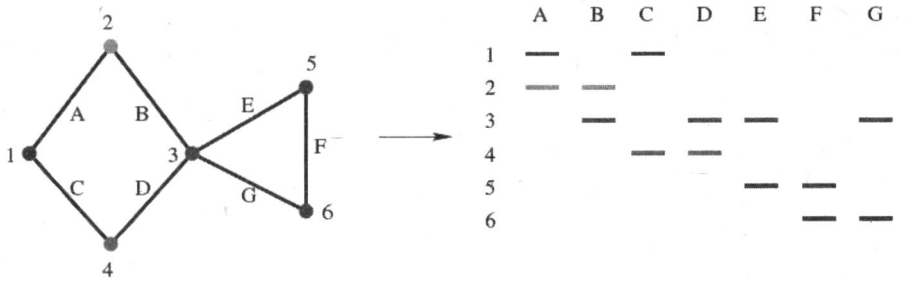



**Рис. 11.5.** Сведение задачи о независимом множестве к задаче календарного планирования (вершины обозначены цифрами, а ребра — буквами)

## 11.3.3. Задача о клике

Социальной кликой (clique) является группа друзей, которые вместе проводят время. В такой группе все знают всех. В теории графов *кликой* называется полный подграф, в котором каждая пара вершин соединена ребром. Клики являются наиболее плотными подграфами. Формальная постановка задачи о клике имеет такой вид:

**Задача**. Найти максимальную клику.

**Вход**. Граф $G = (V, E)$ и целое число $k \leq |V|$.

**Выход**. Содержит ли граф клику из $k$ вершин, т. е. существует ли такое подмножество вершин $S$, где $|S| = k$, для которого каждая пара вершин в $S$ определяет ребро в $G$?

Можно ожидать, что граф дружеских отношений будет содержать большие клики, соответствующие отношениям в семьях, на работе, между соседями, в церковных приходах, учебных заведениях и т. п. Применение клик рассматривается более подробно в *разд. 19.1*.

В случае с задачей о независимом множестве нам нужно было найти подмножество вершин $S$, не являющихся концами общих ребер. Задача о клике отличается от этой задачи тем, что любая пара вершин должна быть соединена ребром. Для сведения этих задач мы выполняем операцию *дополнения* графа — т. е. производим обмен ролями между фактом наличия ребра и фактом его отсутствия (листинг 11.7). На рис. ЦВ-11.6, *а* показан пример графа, содержащего клику из четырех вершин, а на рис. ЦВ-11.6, *б* — соответствующее независимое множество, создающее двухвершинную клику в дополнении графа.

**Листинг 11.7. Сведение задачи о независимом множестве к задаче о клике**

```
IndependentSet(G, k)
      Создаем граф G' = (V', E'), где V = V, и
         For all (i, j) ∈ E, add (i, j) to E'
      Возвращаем Clique(G', k)
```

Два сведения, рассмотренные в *разд. 11.3.2* и *11.3.3*, предоставляют цепочку, связывающую вместе три разные задачи. Сложность задачи о клике следует из сложности

задачи о независимом множестве, которая, в свою очередь, следует из сложности задачи о вершинном покрытии. Выстраивая сведения в цепочку, мы связываем вместе пары задач в соответствии с импликацией сложности. Нашу работу можно считать выполненной, когда все эти цепочки начинаются с одной задачи «Дуэйн Дуглас Джонсон», заведомо считающейся сложной. В такой цепочке первым звеном является задача выполнимости.

## 11.4. Задача выполнимости булевых формул

Для демонстрации сложности задач с помощью сведений нам нужно начать с одной заведомо сложной задачи. Самой сложной изо всех NP-полных задач является логическая задача, называемая задачей *выполнимости* (satisfiability) булевых формул.

Далее приводится формальная постановка этой задачи.

**Задача**. Задача выполнимости булевых формул.

**Вход**. Набор булевых переменных $V$ и набор логических дизъюнкций (clause) $C$ над $V$.

**Выход**. Существует ли выполнимый набор значений истинности для дизъюнкций из $C$ — т. е. способ присвоить каждой из переменных $\{v_1, ..., v_n\}$ значение *истина* или *ложь* таким образом, чтобы каждая дизъюнкция содержала, по крайней мере, один истинный литерал?

Для ясности приведем два примера. Рассмотрим набор $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ над булевыми переменными $V = \{v_1, v_2\}$. Символом $\overline{v}_i$ обозначается дополнение переменной $v_i$ поскольку $\overline{v}_i$ означает НЕ-$v_i$ Это позволяет нам обеспечить выполнимость дизъюнкции, содержащей $v_i$, если $v_i$ = *истина*, или дизъюнкции, содержащей $\overline{v}_i$, если $v_i$ = *ложь*. Следовательно, выполнимость некоторого набора дизъюнкций включает в себя принятие последовательности из $n$ решений *истина* или *ложь* в попытке найти такие булевы значения, при которых выполняются все дизъюнкции.

Множество дизъюнкций $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ соответствует логическому выражению:

$$(v_1 \vee \overline{v}_2) \wedge (\overline{v}_1 \vee v_2)$$

и его можно выполнить, присвоив значения $v_1 = v_2$ = *истина* или $v_1 = v_2$ = *ложь*.

Рассмотрим теперь набор дизъюнкций $\{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$. В этом случае не существует удовлетворительного набора значений, поскольку для выполнения третьей дизъюнкции значение переменной $v_1$ должно быть *ложь*, откуда следует, что для выполнения второй дизъюнкции значение переменной $v_2$ должно быть *ложь*, но тогда не выполняется первое выражение. И сколько бы мы ни старались, получить требуемую выполнимость нам не удастся.

По ряду укоренившихся в общественном сознании, а также технических причин, существует устоявшееся мнение, что задача выполнимости булевых формул является сложной и для ее решения не существует алгоритмов с полиномиальным временем исполнения в худшем случае. Буквально все эксперты в области разработки алгоритмов

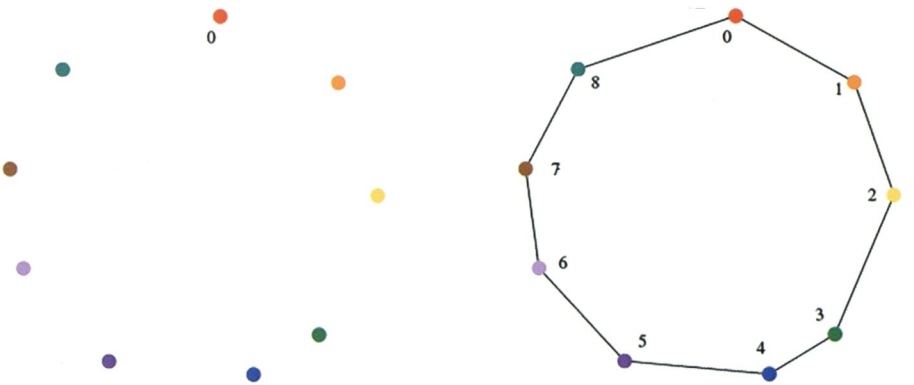



**Рис. ЦВ-1.2.** Случай удачного входного экземпляра для эвристического алгоритма ближайшего соседа. Цветная разметка точек, соответствующая порядку следования цветов радуги (от красного к фиолетовому), отражает порядок перемещения манипулятора

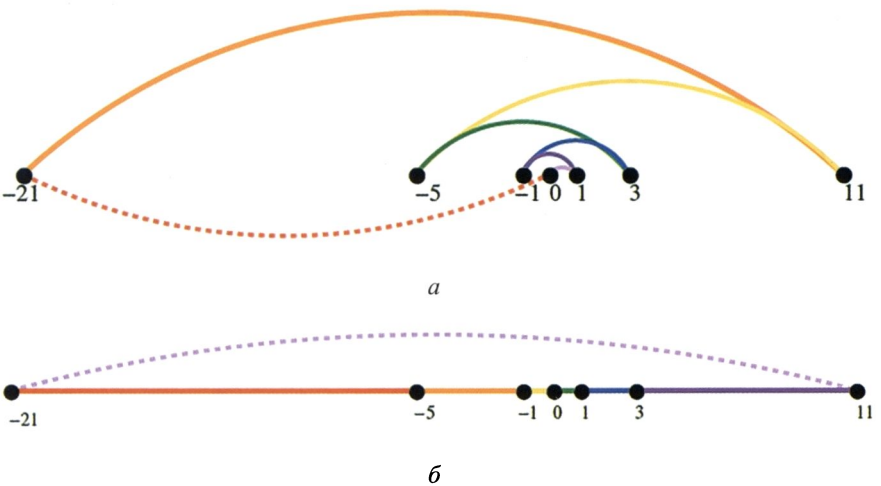



**Рис. ЦВ-1.3.** Пример неудачного входного экземпляра для эвристического алгоритма ближайшего соседа (*а*) и оптимальное решение (*б*). Цвета последовательности перемещений манипулятора соответствуют порядку цветов спектра

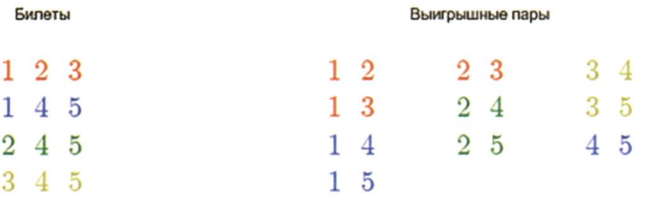



**Рис. ЦВ-1.11.** Покрытие всех пар множества {1, 2, 3, 4, 5} номерами билетов {1, 2, 3}, {1, 4, 5}, {2, 4, 5}, {3, 4, 5}. Цвет пары представляет покрывающий билет

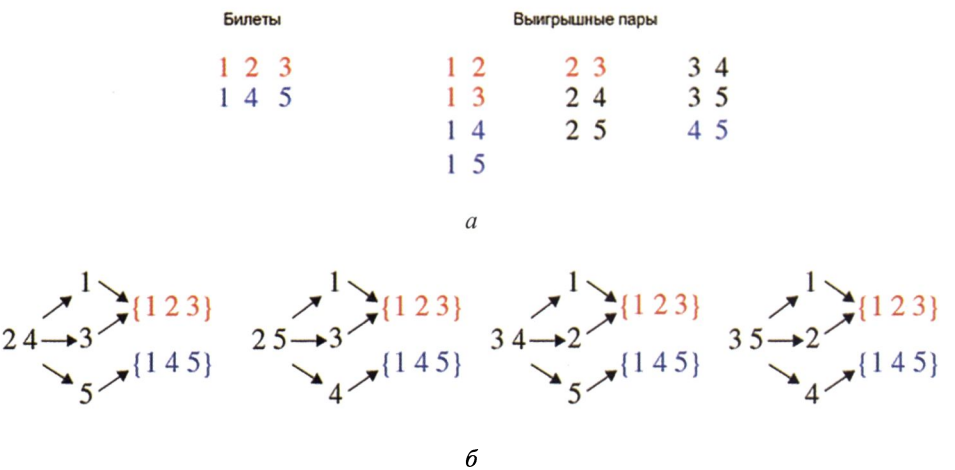



**Рис. ЦВ-1.12.** *а* — гарантия выигрышной комбинации из двух номеров множества {l, 2, 3, 4, 5} при использовании только комбинации номеров {l, 2, 3} и {l, 4, 5};
*б* — все отсутствующие пары подразумевают пару, входящую в покрытие при разложении. Цвет пары представляет покрывающий билет

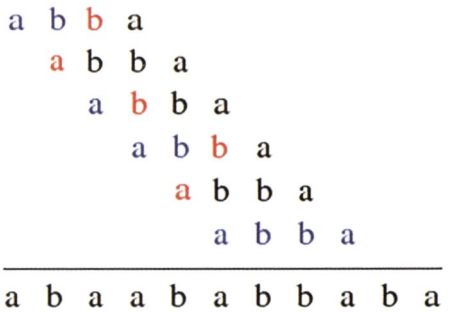



**Рис. ЦВ-2.5.** Пример поиска подстроки *abba* в тексте *abaababbaba*. Буквы синего цвета представляют совпадения с требуемой подстрокой, а красного — несовпадения. Поиск прекращается при обнаружении первого полного совпадения

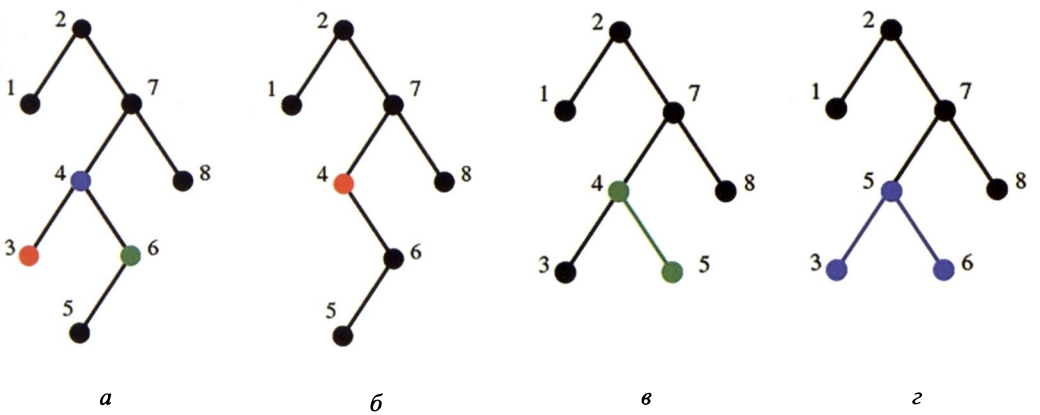



**Рис. ЦВ-3.5.** *а* — исходное дерево; *б* — удаление узла дерева без потомков (3); *в* — удаление узла дерева с одним потомком (6); *г* — удаление узла дерева с двумя потомками (4). Затронутые в результате удаления узлы обозначены разными цветами

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 34 | 5 | 55 | 21 | | 2 | | 3 | 8 | 13 |

**Рис. ЦВ-3.10.** Разрешение коллизий методом открытой адресации и последовательного исследования после вставки первых восьми чисел Фибоначчи в возрастающем порядке посредством функции хэширования $H(x) = (2x + 1) \bmod 10$. Элементы, выделенные красным, были вставлены в первую свободную ячейку после недоступной желаемой ячейки

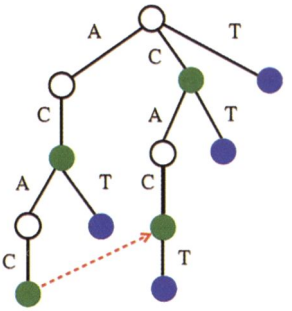
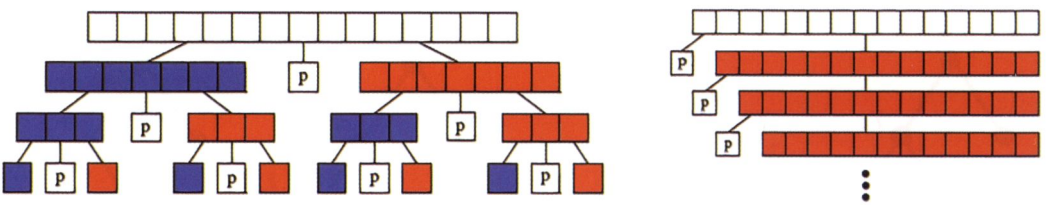



**Рис. ЦВ-3.12.** Суффиксное дерево строк *ACAC* и *CACT* с указателем на суффикс строки *ACAC*. Узлы зеленого цвета соответствуют суффиксам *ACAC*, а синего цвета — суффиксам *CACT*


P
P
P

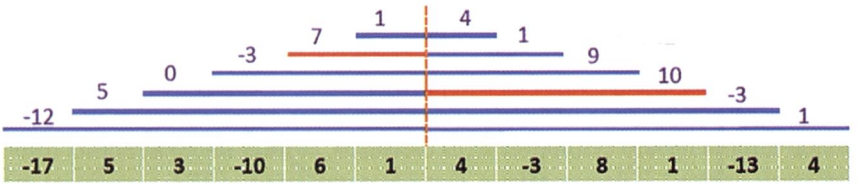



**Рис. ЦВ-5.3.** Поддиапазон с наибольшей суммой значений будет находиться или полностью в левой, или полностью в правой части, или же, как в этом случае, будет суммой наибольших поддиапазонов слева и справа от центра всего диапазона

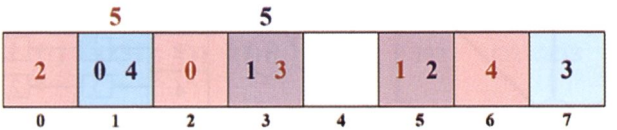



**Рис. ЦВ-6.10.** Хэширование целых чисел 0, 1, 2, 3 и 4 в фильтре Блума с n = 8 с использованием функции хэширования $h_1(x) = 2x + 1$ (синий цвет) и $h_2(x) = 3x + 2$ (красный цвет). При поиске $x = 5$ будет возвращен ложный положительный результат, поскольку соответствующие два бита были установлены другими элементами

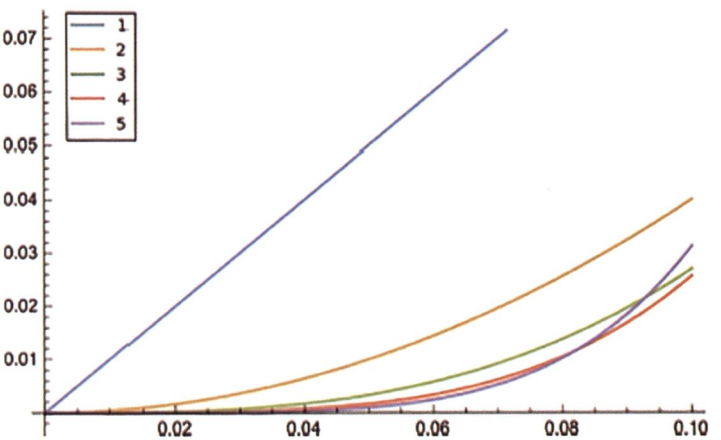



**Рис. ЦВ-6.11.** Вероятность ошибки фильтра Блума в зависимости от нагрузки для значений $k$ от 1 до 5. Выбрав правильное значение $k$ для заданной нагрузки, можно значительно снизить частоту ошибок ложных коллизий, не увеличивая при этом размер таблицы




**Рис. ЦВ-6.15.** Функция хэширования по алгоритму Рабина — Карпа *H*(*s, j*) выдает различные хэш-коды для разных подстрок (отображены синим цветом), тогда как менее мощная функция хэширования, которая просто суммирует коды символов, создает много коллизий (отображены фиолетовым цветом). В нашем случае длина искомой строки (*BBA*) составляет m = 3, а коды символов равны *A* = 0 и *B* = 1

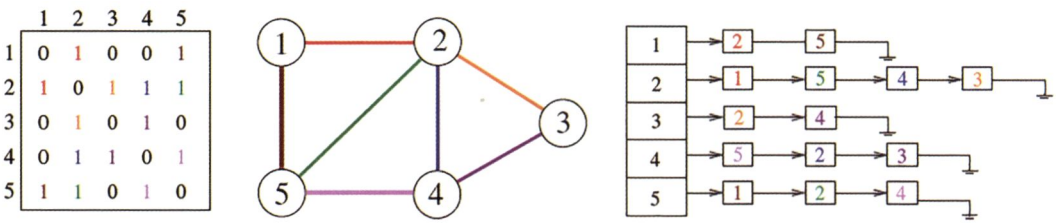
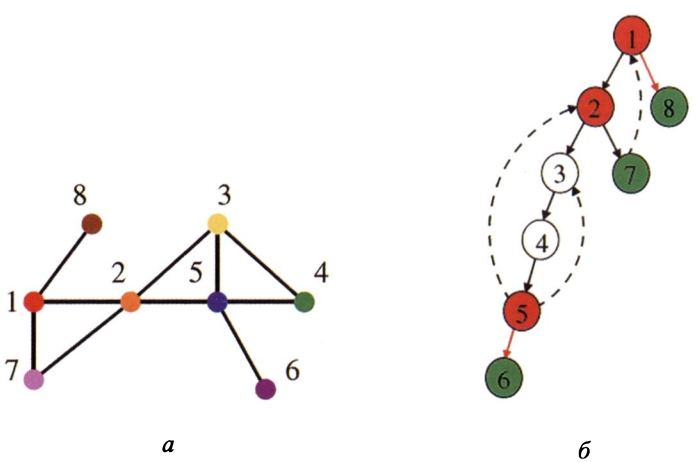
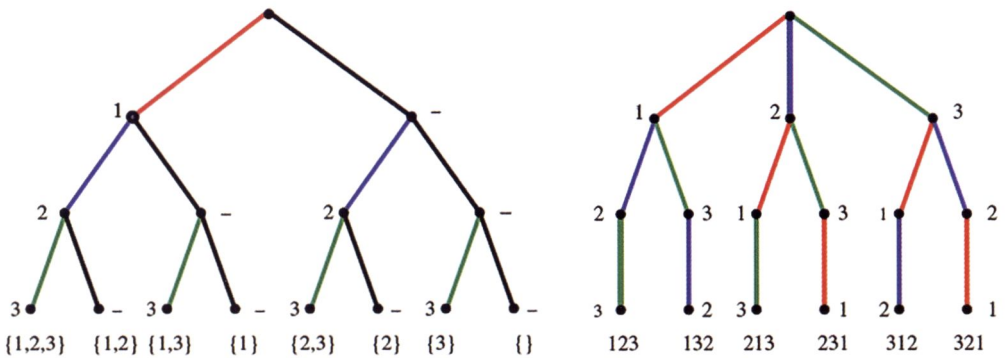



**Рис. ЦВ-7.4.** Представление графа (*в центре*) матрицей смежности (*слева*) и списком смежности (*справа*). Конкретные ребра обозначены соответствующими цветами



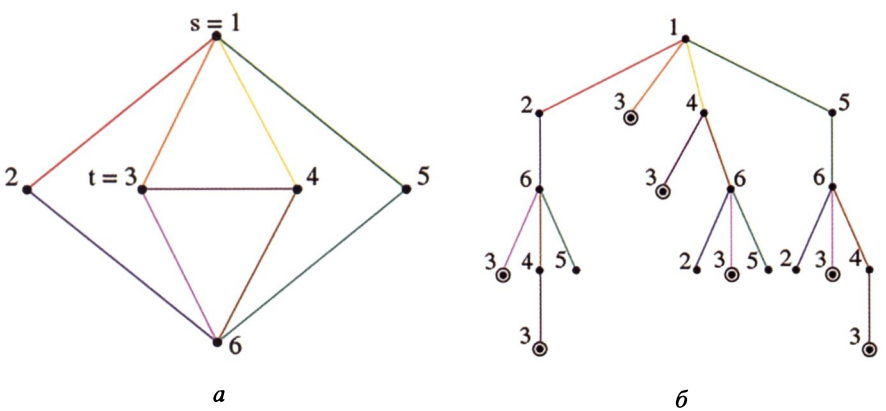

*а* *б*

**Рис. ЦВ-7.11.** Связный граф (*а*) телефонной сети и его дерево обхода в глубину (*б*), содержащее шарнирные вершины 1, 2 и 5. Обратные ребра не дают вершинам 3 и 4 быть шарнирами, а зеленые вершины 6, 7 и 8 не могут быть шарнирами, потому что они листья (концевые узлы) дерева обхода графа в глубину. Красные ребра (1, 8) и (5, 6) являются мостовыми ребрами, удаление которых превращает граф в несвязный




*а* *б*

**Рис. ЦВ-9.1.** Дерево поиска с перечислением всех подмножеств (*а*) и перестановок (*б*) множества {1, 2, 3}. Цвет каждого ребра дерева отображает вставку элемента в частичное решение




*а* *б*

**Рис. ЦВ-9.2.** Граф (*а*) и дерево поиска со всеми путями между вершинами *s* и *t* (*б*). Цвет каждого ребра дерева поиска сопоставлен цвету соответствующего ребра графа

| | $T$ | | y | o | u | - | s | h | o | u | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | pos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| : | | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| t: | 1 | **1** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| h: | 2 | 2 | **2** | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | 9 |
| o: | 3 | 3 | 3 | **2** | 3 | 4 | 5 | 6 | 5 | 6 | 7 | 8 |
| u: | 4 | 4 | 4 | 3 | **2** | 3 | 4 | 5 | 6 | 5 | 6 | 7 |
| -: | 5 | 5 | 5 | 4 | 3 | **2** | 3 | 4 | 5 | 6 | 6 | 7 |
| s: | 6 | 6 | 6 | 5 | 4 | 3 | **2** | 3 | 4 | 5 | 6 | 7 |
| h: | 7 | 7 | 7 | 6 | 5 | 4 | 3 | **2** | **3** | 4 | 5 | 6 |
| a: | 8 | 8 | 8 | 7 | 6 | 5 | 4 | 3 | 3 | **4** | 5 | 6 |
| l: | 9 | 9 | 9 | 8 | 7 | 6 | 5 | 4 | 4 | 4 | **4** | 5 |
| t: | 10 | 10 | 10 | 9 | 8 | 7 | 6 | 5 | 5 | 5 | 5 | **5** |

**Рис. ЦВ-10.5.** Пример матрицы стоимости преобразования строки $P$ = thou shalt в строку $T$ = you should за пять шагов. Значения на пути оптимального выравнивания выделены подчеркиванием. Синим цветом обозначены вставки, зеленым — удаления, а красным — замены

| | $T$ | | y | o | u | - | s | h | o | u | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$ | pos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | 0 | **-1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| t: | 1 | **2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| h: | 2 | 2 | **0** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| o: | 3 | 2 | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| u: | 4 | 2 | 0 | 2 | **0** | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| -: | 5 | 2 | 0 | 2 | 2 | **0** | 1 | 1 | 1 | 1 | 0 | 0 |
| s: | 6 | 2 | 0 | 2 | 2 | 2 | **0** | 1 | 1 | 1 | 1 | 0 |
| h: | 7 | 2 | 0 | 2 | 2 | 2 | 2 | **0** | **1** | 1 | 1 | 1 |
| a: | 8 | 2 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | **0** | 0 | 0 |
| l: | 9 | 2 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | **0** | 1 |
| t: | 10 | 2 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | **0** |

**Рис. ЦВ-10.6.** Родительская матрица для вычисления расстояния редактирования. Значения на пути оптимального выравнивания выделены подчеркиванием. Синим цветом обозначены вставки, зеленым — удаления, а красным — замены

| $i$ | $s_i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1 | 1 | -1 | 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 2 | 2 | -1 | -1 | 0 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 3 | 4 | -1 | -1 | -1 | -1 | 0 | 1 | 2 | 3 | -1 | -1 | -1 | -1 |
| 4 | 8 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 | 1 | 2 | 3 |

**Рис. ЦВ-10.10.** Массив родительских элементов для решения, равного 11. Красным цветом выделены ячейки, обнаруженные на обратном проходе в процессе восстановления решения

| $M$ | $k$ | | | | $D$ | $k$ | | |
|---|---|---|---|---|---|---|---|---|
| $s$ | 1 | 2 | 3 | | $s$ | 1 | 2 | 3 |
| 1 | 1 | 1 | 1 | | 1 | – | – | – |
| 1 | 2 | 1 | 1 | | 1 | – | 1 | 1 |
| 1 | 3 | 2 | 1 | | 1 | – | 1 | 2 |
| 1 | 4 | 2 | 2 | | 1 | – | 2 | 2 |
| 1 | 5 | 3 | 2 | | 1 | – | 2 | 3 |
| 1 | 6 | 3 | 2 | | 1 | – | 3 | 4 |
| 1 | 7 | 4 | 3 | | 1 | – | 3 | 4 |
| 1 | 8 | 4 | 3 | | 1 | – | 4 | 5 |
| 1 | 9 | 5 | 3 | | 1 | – | 4 | 6 |

*а*

| $M$ | $k$ | | | | $D$ | $k$ | | |
|---|---|---|---|---|---|---|---|---|
| $s$ | 1 | 2 | 3 | | $s$ | 1 | 2 | 3 |
| 1 | 1 | 1 | 1 | | 1 | – | – | – |
| 2 | 3 | 2 | 2 | | 2 | – | 1 | 1 |
| 3 | 6 | 3 | 3 | | 3 | – | 2 | 2 |
| 4 | 10 | 6 | 4 | | 4 | – | 3 | 3 |
| 5 | 15 | 9 | 6 | | 5 | – | 3 | 4 |
| 6 | 21 | 11 | 9 | | 6 | – | 4 | 5 |
| 7 | 28 | 15 | 11 | | 7 | – | 5 | 6 |
| 8 | 36 | 21 | 15 | | 8 | – | 5 | 6 |
| 9 | 45 | 24 | 17 | | 9 | – | 6 | 7 |

*б*

**Рис. ЦВ-10.11.** Матрицы динамического программирования для двух входных экземпляров задачи упорядоченного разбиения: *а* — разбиение множества (1, 1, 1, 1, 1, 1, 1, 1, 1) на подмножества ((1, 1, 1), (1, 1, 1), (1, 1, 1)) и *б* — множества (1, 2, 3, 4, 5, 6, 7, 8, 9) на подмножества ((1, 2, 3, 4, 5), (6, 7), (8, 9)) (справа). Префиксные суммы выделены красным цветом, а позиции разделителей оптимального решения — синим

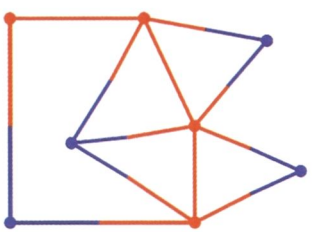
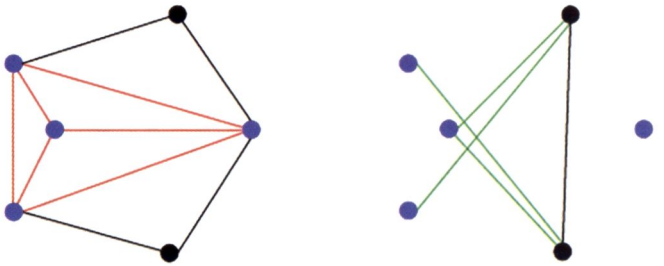
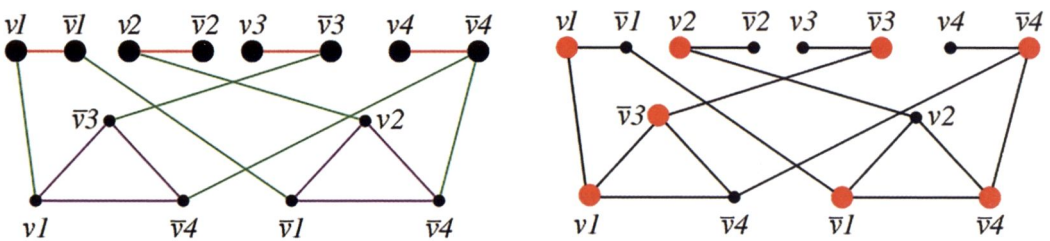


**Рис. ЦВ-11.4.** Обозначенные красным цветом вершины составляют вершинное покрытие, поэтому остальные вершины должны составлять независимое множество



*а* *б*

**Рис. ЦВ-11.6.** *а* — пример небольшого графа с четырехвершинной кликой (выделена синим цветом); *б* — соответствующее независимое множество (выделено черным цветом), создающее двухвершинную клику в дополнении графа




*а* *б*

**Рис. ЦВ-11.7.** *а* — сведение 3-SAT экземпляра $\{\{v_1, \overline{v}_3, \overline{v}_4\}, \{\overline{v}_1, v_2, \overline{v}_4\}\}$ задачи выполнимости к задаче о вершинном покрытии; *б* — красные вершины определяют минимальное вершинное покрытие, следовательно, красные вершины верхних переменных определяют выполняющий набор значений истинности

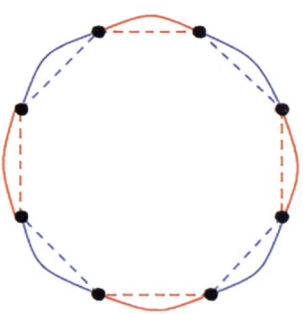


**Рис. ЦВ-12.5.** Любой маршрут коммивояжера в графе с четным количеством вершин, удовлетворяющем условиям аксиомы треугольника, можно разделить на красные и синие паросочетания, стоимость каждого из которых должна быть не более чем половина стоимости всего маршрута

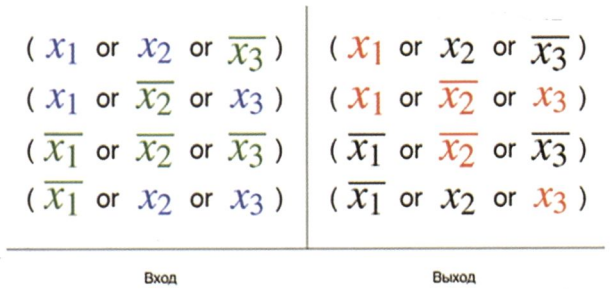



**Рис. ЦВ-17.11.** Задача выполнимости

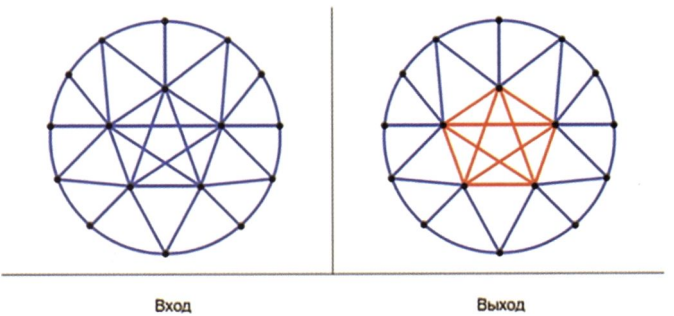



**Рис. ЦВ-19.1.** Клика

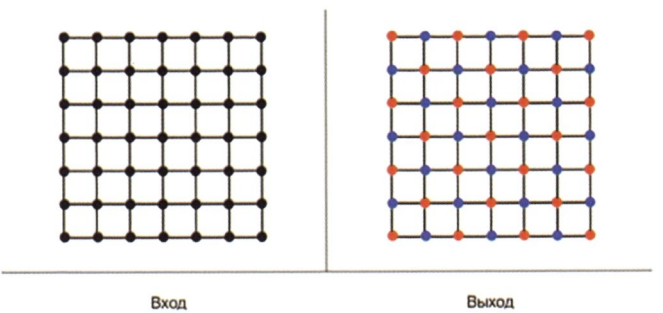



**Рис. ЦВ-19.2.** Независимое множество

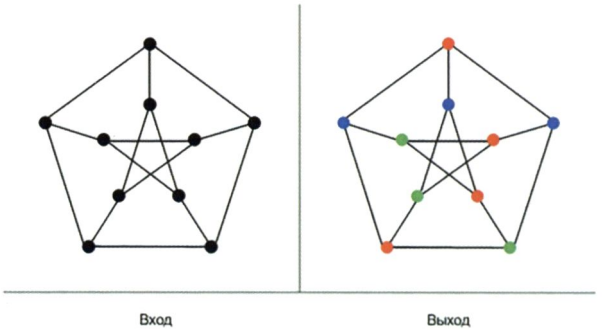



**Рис. ЦВ-19.8.** Вершинная раскраска

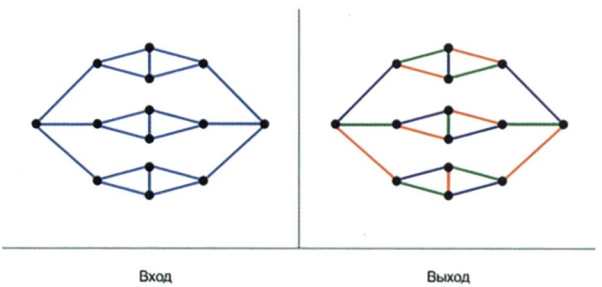



**Рис. ЦВ-19.9.** Реберная раскраска




**Рис. ЦВ-21.4**. Поиск в строке по образцу

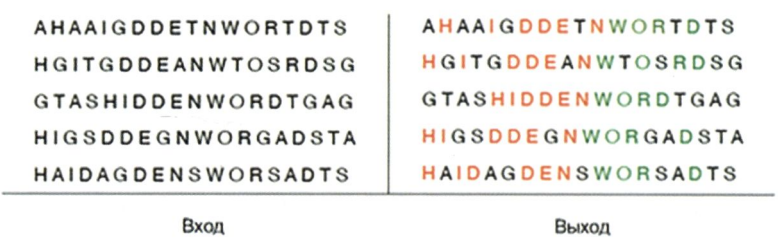



**Рис. ЦВ-21.9.** Поиск максимальной общей подстроки

(и бесчисленное множество просто хороших профессионалов) пытались непосредственно или косвенно разработать эффективный алгоритм для выяснения выполнимости любого набора дизъюнкций. Все они потерпели неудачу. Кроме этого, было доказано, что если бы существовал эффективный алгоритм для решения задачи выполнимости, то в области вычислительной сложности были бы возможными многие странные и невероятные ситуации.

Доказательство, что некая задача такая сложная, как задача выполнимости, означает, что она действительно сложная. Дополнительную информацию по задаче выполнимости и ее применении вы найдете в *разд. 17.10*.

### 11.4.1. Задача выполнимости в 3-конъюнктивной нормальной форме (задача 3-SAT)

Занимаемое задачей выполнимости первое место среди NP-полных задач обусловило тот факт, что ее трудно решить для наихудшего случая. Но некоторые экземпляры частных случаев задачи не обязательно такие сложные. Допустим, что каждая дизъюнкция содержит ровно один литерал — скажем: $\{v_i\}$ или $\{\overline{v}_j\}$. Ясно, что присвоить литералу значение, которое бы удовлетворяло такую дизъюнкцию, можно только одним способом: чтобы иметь какой-либо шанс удовлетворения полного множества дизъюнкций, $v_i$ нужно присвоить значение *истина*, а $v_j$ — *ложь*. Таким образом, этот набор будет невыполнимым только в том случае, если имеются две дизъюнкции, которые прямо противоречат одна другой, — например, $C = \{\{v_1\},\{\overline{v}_1\}\}$.

Поскольку определить, поддаются ли выполнению наборы дизъюнкций, каждая из которых имеет только один литерал, не представляет особого труда, нас интересуют чуть более длинные дизъюнкции. Сколько требуется литералов в каждой дизъюнкции, чтобы превратить задачу из решаемой за полиномиальное время в труднорешаемую? Это происходит, когда каждая дизъюнкция содержит три литерала. Формальное определение выглядит таким образом:

**Задача**. Задача выполнимости в 3-конъюнктивной нормальной форме (3-SAT).

**Вход.** Коллекция дизъюнкций $C$, каждая из которых содержит ровно 3 литерала, над набором булевых переменных $V$.

**Выход**. Существует ли набор значений истинности для переменных $V$, при котором выполняется каждая дизъюнкция?

Поскольку задача 3-SAT представляет собой частный случай задачи выполнимости, из сложности этой задачи неявно следует сложность общей задачи выполнимости. Обратное утверждение неверно, поскольку сложность общей задачи выполнимости могла бы предположительно зависеть от длин дизъюнкций. Но сложность задачи 3-SAT можно показать с помощью сведения, которое преобразует каждый экземпляр задачи выполнимости в экземпляр задачи 3-SAT, не нарушая при этом его выполнимость.

Такое сведение преобразует каждую дизъюнкцию отдельно — в зависимости от ее длины — путем добавления новых дизъюнкций и булевых переменных. Допустим, что дизъюнкция $C_i$ содержит $k$ литералов:

- $k = 1$, скажем, $C_i = \{z_1\}$.

  Создадим две новые переменные $v_1$, $v_2$ и четыре новые дизъюнкции по 3 литерала каждая: $\{v_1, v_2, z_1\}, \{v_1, \overline{v}_2, z_1\}, \{\overline{v}_1, v_2, z_1\}$ и $\{\overline{v}_1, \overline{v}_2, z_1\}$. Обратите внимание, что все эти четыре дизъюнкции могут быть одновременно выполнены только в том случае, если значение $z_1$ = *истина*, что также означает выполнимость первоначальной дизъюнкции $C_i$.

- $k = 2$, скажем, $C_i = \{z_1, z_2\}$.

  Создадим одну новую переменную $v_1$ и две новые дизъюнкции по 3 литерала: $\{v_1, z_2, z_1\}$ и $\{\overline{v}_1, z_2, z_1\}$. Так же как и в предыдущем случае, эти две дизъюнкции могут быть одновременно выполнены только в том случае, если, по крайней мере, значение одной из переменных $z_1$ и $z_2$ равно *истина*, что также означает выполнимость $C_i$;

- $k = 3$, скажем, $C_i = \{z_1, z_2, z_3\}$.

  Дизъюнкция $C_i$ копируется в экземпляр задачи 3-SAT без изменений: $\{z_1, z_2, z_3\}$.

- $k > 3$, скажем, $C_i = \{z_1, z_2, \ldots, z_n\}$.

  Здесь мы последовательно создаем $k-3$ новых переменных и $k-2$ новых дизъюнкций, где $C_{i,1} = \{z_1, z_2, \overline{v}_{i,1}\}, C_{i,j} = \{v_{i,j-1}, z_{j+1}, \overline{v}_{i,j}\}$ для $2 \le j \le k-3$ и $C_{i,k-2} = \{v_{i,k-3}, z_{k-1}, z_k\}$.

  Этот случай лучше всего иллюстрируется на примере. Так, дизъюнкция:

$$C_i = \{z_1, z_2, z_3, z_4, z_5, z_6\}$$

  преобразовывается в следующий набор из четырех 3-литеральных дизъюнкций с тремя булевыми переменными $v_{i,1}$, $v_{i,2}$ и $v_{i,3}$:

$$\{\{z_1, z_2, \overline{v}_{i,1}\}, \{v_{i,1}, z_3, \overline{v}_{i,2}\}, \{v_{i,2}, z_4, \overline{v}_{i,3}\}\{v_{i3}, z_5, z_6\}\}.$$

Наиболее сложным является случай с длинными дизъюнкциями. Если ни один из первоначальных литералов не имеет значение *истина*, то тогда не будет достаточного количества новых свободных переменных, чтобы можно было выполнить все новые дизъюнкции. Дизъюнкцию $C_{i,1}$ можно выполнить, присвоив литералу $v_{i,1}$ значение *ложь*, но это заставит литерал $v_{i,2}$ принять значение *ложь*, и т. д., пока не окажется, что дизъюнкция $C_{i,k-2}$ не может быть выполнена. Но если значение любого литерала $z_i$ равно *истина*, то у нас имеется $k-3$ свободных переменных и $k-3$ оставшихся 3-литеральных дизъюнкций, так что все они могут быть выполнены.

Это преобразование выполняется за время $O(n+c)$, если экземпляр задачи выполнимости задачи SAT имеет $c$ дизъюнкций и $n$ литералов. Так как любое решение исходного экземпляра задачи SAT также является решением созданного нами экземпляра задачи 3-SAT, и наоборот, то преобразованная задача эквивалентна первоначальной.

Обратите внимание, что незначительная модификация этой конструкции может быть использована для доказательсta того, что задачи 4-SAT, 5-SAT и, вообще, $(k \ge 3)$-SAT также являются NP-полными. Но эта конструкция неприменима для задачи 2-SAT, поскольку у нас нет способа заполнить цепочку дизъюнкций. Однако для решения задачи

2-SAT за линейное время можно использовать алгоритм обхода в ширину на соответствующем графе. Подробности приведены в *разд. 17.10*.

# 11.5. Нестандартные сведения задачи SAT

Поскольку известно, что как общая задача выполнимости, так и задача выполнимости 3-SAT являются труднорешаемыми, в сведениях можно использовать любую из них. Обычно для этого лучше подходит задача 3-SAT, поскольку с ней проще работать. Чтобы предоставить вам дополнительные примеры и расширения нашего списка известных труднорешаемых задач, мы рассмотрим еще два сложных сведения.

Многие сведения весьма сложны, поскольку мы, по сути, программируем одну задачу на языке другой задачи, значительно отличающейся от первой. Больше всего путаницы возникает при выборе направления сведения. Запомните, что требуется преобразовать *любой* экземпляр заведомо NP-полной задачи (Bandersnatch) в экземпляр задачи, которая нас действительно интересует (Bo-billy). Если выполнить сведение в обратном направлении, то мы получим лишь медленный способ решения интересующей нас задачи в виде процедуры с экспоненциальным временем исполнения. Это может сбить с толку начинающего алгоритмиста, т. к. указанное направление сведения кажется обратным требуемому. Обязательно разберитесь с правильным направлением сведения сейчас, и обращайтесь к предлагаемому здесь материалу в случае затруднений в этом вопросе.

## 11.5.1. Вершинное покрытие

Алгоритмическая теория графов изобилует сложными задачами. Прототипичной NP-полной задачей теории графов является задача поиска вершинного покрытия, которая была определена в *разд. 11.3.2* таким образом:

**Задача**. Найти вершинное покрытие.

**Вход.** Граф $G = (V, E)$ и целое число $k \leq |V|$.

**Выход.** Существует ли такое подмножество $S$, содержащее самое большее $k$ вершин, для которого, по крайней мере, одна вершина каждого ребра $e \in E$ является элементом $S$?

Доказать сложность задачи о вершинном покрытии труднее, чем сложность задач в ранее рассмотренных сведениях, из-за кажущегося значительного различия структур соответствующих задач. Сведение задачи 3-SAT к задаче о вершинном покрытии требует создания графа $G$ и границы $k$ из переменных и дизъюнкций экземпляра задачи выполнимости.

Подойдем к этом следующим образом. Сначала преобразуем переменные задачи 3-SAT. Для каждой булевой переменной $v_i$ мы создадим две вершины: $v_i$ и $\overline{v}_i$, которые соединяются ребром. Лишь для покрытия этих ребер нам потребуется, по крайней мере, $n$ из этих $2n$ вершин, поскольку для пары ребер нам нужна как минимум одна вершина.

Далее мы преобразуем дизъюнкции задачи 3-SAT. Для каждой из $c$ дизъюнкций мы создадим три новые вершины — по одной для каждого литерала в дизъюнкции. Эти три вершины будут соединены таким образом, чтобы создать для каждой дизъюнкции

треугольник. В любое вершинное покрытие этих треугольников должны быть включены, по крайней мере, две вершины каждого треугольника, при этом общее число вершин покрытия будет равным $2c$.

Наконец, мы соединим вместе эти два набора компонентов. Каждый литерал в вершинном компоненте соединяется с вершинами в компонентах дизъюнкций (треугольниках), разделяющих этот литерал. Таким образом, из экземпляра задачи 3-SAT с $n$ переменными и $c$ дизъюнкциями создается граф, имеющий $2n + 3c$ вершин. Полное сведение для задачи 3-SAT $\{\{v_1, \overline{v}_3, \overline{v}_4\}, \{\overline{v}_1, v_2, \overline{v}_4\}\}$ показано на рис. ЦВ-11.7.

Этот граф создан таким образом, чтобы вершинное покрытие размером $n + 2c$ было возможным тогда и только тогда, когда первоначальное выражение является выполнимым. Согласно ранее приведенному анализу любое вершинное покрытие должно содержать, по крайней мере, $n + 2c$ вершин. Для доказательства правильности нашего сведения нам нужно продемонстрировать, что верны следующие утверждения.

- *Каждый выполняющий набор значений истинности дает вершинное покрытие размером $n + 2c$.*

  Для имеющегося выполняющего набора значений истинности выберите для дизъюнкций в качестве членов вершинного покрытия $n$ вершин из вершинных компонентов, которые соответствуют истинным литералам. Так как это определяет выполняющий набор значений истинности, то истинный литерал из каждой дизъюнкции должен покрывать, по крайней мере, одно из трех поперечных ребер, соединяющих каждую вершину треугольника с компонентом вершин. Соответственно, выбирая две другие вершины каждого треугольника дизъюнкции, мы также выбираем все оставшиеся поперечные ребра.

- *Каждое вершинное покрытие размером $n + 2c$ дает выполняющий набор значений истинности.*

  В любом вершинном покрытии $C$ размером $n + 2c$ ровно $n$ вершин должны принадлежать вершинным компонентам. Пусть эти вершины первого этапа определяют набор значений истинности, в то время как остальные $2c$ вершин покрытия распределяются по две вершины на каждый компонент дизъюнкции, ибо в противном случае ребро компонента дизъюнкции было бы непокрытым. Эти вершины компонентов дизъюнкции могут покрывать только два из трех соединяющих ребер для каждой дизъюнкции. Поэтому если $C$ обеспечивает вершинное покрытие, то, по крайней мере, одно соединяющее ребро в каждой дизъюнкции должно быть покрыто вершинами первого этапа, а это означает, что соответствующий набор значений истинности является выполняющим для всех дизъюнкций.

Это доказательство сложности задачи о вершинном покрытии, объединенное в цепочку со сведениями задачи о клике и задачи о независимом множестве из *разд. 11.3.2*, составляет библиотеку сложных задач теории графов, которые мы можем использовать для облегчения доказательства сложности других задач.

### Подведение итогов

Небольшого набора NP-полных задач (3-SAT, вершинное покрытие, разделение множества целых чисел и гамильтонов цикл) вполне достаточно для доказательства сложности большинства других сложных задач.

## 11.5.2. Целочисленное программирование

Рассматриваемое в *разд. 16.6* целочисленное программирование представляет собой фундаментальную задачу комбинаторной оптимизации. Ее лучше всего рассматривать, как задачу линейного программирования, в которой переменные могут принимать только целочисленные значения (а не вещественные). Формальная постановка задачи имеет следующую форму:

**Задача**. Целочисленное программирование.

**Вход**. Набор целочисленных переменных $V$, набор неравенств над $V$, линейная функция максимизации $f(V)$ и целое число $B$.

**Выход.** Существует ли набор целочисленных значений переменных $V$, для которого выполняются все неравенства и $f(V) \geq B$?

Рассмотрим два примера. Допустим, что

$$V_1 \geq 1, V_2 \geq 0,$$
$$V_1 + V_2 \leq 3,$$
$$f(V) : 2V_2, B = 3.$$

Решением для такого входного экземпляра было бы $V_1 = 1$, $V_2 = 2$. Обратите внимание, что это решение соблюдает целостность и дает целевое значение $f(V) = 4 \geq B$. Но не все задачи обладают реализуемыми решениями. Рассмотрим следующий входной экземпляр для этой же задачи:

$$V_1 \geq 1, V_2 \geq 0,$$
$$V_1 + V_2 \leq 3,$$
$$f(V) : 2V_2, B = 5.$$

При таких ограничениях максимальное значение функции $f(V)$ равно $2 \times 2 = 4$, поэтому соответствующая задача разрешимости не имеет решения.

Для доказательства труднорешаемости задачи целочисленного программирования мы выполним сведение от общей задачей выполнимости. Как правило, использование задачи 3-SAT облегчает сведение, и в нашем конкретном случае с тем же успехом можно было бы воспользоваться этим подходом точно таким же образом.

В каком направлении должно выполняться сведение? Мы хотим доказать труднорешаемость задачи целочисленного программирования и знаем, что задача выполнимости является труднорешаемой. Если задачу выполнимости можно было бы решить, используя целочисленное программирование, и задача целочисленного программирования не была бы трудной, то и задача выполнимости не была бы трудной. Отсюда следует направление сведения: задачу выполнимости (Bandersnatch) нужно преобразовать в задачу целочисленного программирования (Bo-billy).

Каким должно быть это преобразование? Каждый экземпляр задачи выполнимости содержит булевы переменные и дизъюнкции. Каждый экземпляр задачи целочисленного программирования содержит целочисленные переменные и ограничения. Имеет смысл сопоставить целочисленные переменные с булевыми и использовать ограничения в той же роли, какую играют дизъюнкции в исходной задаче.

Преобразованная задача целочисленного программирования будет содержать вдвое больше переменных, чем соответствующий экземпляр задачи SAT, — по одной переменной для каждой исходной булевой переменной и для ее дополнения. Для каждой переменной $v_i$ в задаче выполнимости добавляются следующие ограничения:

- чтобы каждая переменная $V_i$ целочисленного программирования принимала только значения 0 или 1, добавляются ограничения $0 \le V_i \le 1$ и $0 \le \overline{V}_i \le 0$. С учетом целочисленности переменных эти значения соответствуют значениям *истина* и *ложь*;
- для гарантии того, что одна и только одна из двух переменных в задаче целочисленного программирования, связанных с этой переменной в задаче выполнимости, имеет значение *истина*, добавляются ограничения $1 \le V_i + \overline{V}_i \le 1$.
- Для каждой дизъюнкции $C_i = \{z_1, ..., z_k\}$ создадим ограничение:

$$Z_1 + ... + Z_k \ge 1.$$

  Чтобы удовлетворить это ограничение, по крайней мере, одному из литералов в каждой дизъюнкции нужно присвоить значение 1, чтобы он соответствовал литералу *истина*. Таким образом, выполнимость этого ограничения эквивалентна выполнимости дизъюнкции.

В нашем случае функция максимизации, а также граница теряют свою актуальность, поскольку мы уже закодировали весь экземпляр задачи выполнимости. Используя $f(V) = V_1$ и $B = 0$, мы гарантируем, что они не будут противоречить никаким значениям переменных, удовлетворяющим всем неравенствам. Очевидно, что это сведение можно выполнить за полиномиальное время. Чтобы установить, что при этом сведении сохраняется правильность ответа, нам нужно подтвердить следующие два обстоятельства:

- *любое решение задачи выполнимости дает решение задачи целочисленного программирования.*

  В любом решении задачи выполнимости литерал *истина* соответствует значению 1 в решении задачи целочисленного программирования, поскольку дизъюнкция выполняется. Следовательно, сумма в каждом неравенстве больше или равна 1;
- *любое решение задачи целочисленного программирования дает решение первоначальной задачи выполнимости.*

  В любом решении этого экземпляра задачи целочисленного программирования всем переменным должно быть присвоено значение 0 или 1. Если $V_i = 1$, тогда литералу $z_i$ присваивается значение *истина*. Если $V_i = 0$, тогда литералу $z_i$ присваивается значение *ложь*. Это законное присваивание значений, которое должно удовлетворять все дизъюнкции.

Так как сведение выполняется в обоих направлениях, то задача целочисленного программирования должна быть труднорешаемой. Обратите внимание на следующие свойства, которые остаются в силе и при доказательстве NP-полноты:

- При этом сведении сохраняется структура задачи. Задача *не решается*, а просто преобразуется в другой формат.
- Возможные экземпляры задачи целочисленного программирования, которые могут получиться в результате такого преобразования, представляют только небольшое

подмножество всех возможных экземпляров этой задачи. Но поскольку экземпляры в этом небольшом подмножестве являются труднорешаемыми, то и общая задача, очевидно, является труднорешаемой.

- Это преобразование отражает суть того, *почему* задачи целочисленного программирования трудны для решения. Оно не имеет никакого отношения к большим коэффициентам или большим диапазонам переменных, поскольку ограничение множества значений до 0 и 1 является достаточным. Оно также не имеет ничего общего с большим количеством переменных в неравенствах. Задача целочисленного программирования является труднорешаемой потому, что труднорешаемой задачей является выполнимость набора ограничений. Внимательное изучение свойств задачи, необходимых для ее сведения, может многое рассказать нам о самой задаче.

## 11.6. Искусство доказательства сложности

Доказательство сложности задач требует определенного мастерства. Однако, приобретя навык, вы обнаружите, что выполнение сведений может быть на удивление простым процессом. Малоизвестной особенностью доказательств NP-полноты является тот факт, что их легче создать, чем объяснить, аналогично тому, как часто бывает легче заново написать код, чем разбираться в старом.

Умение определить, какие задачи могут, скорее всего, быть труднорешаемыми, требует опыта. Самый быстрый способ набраться такого опыта — внимательное изучение примеров в каталоге задач. Незначительное изменение формулировки задачи может сделать полиномиальную задачу NP-полной. Задача поиска кратчайшего пути в графе является легкой, в то время как задача поиска самого длинного пути в графе является сложной. Задача построения маршрута, который проходит по всем ребрам графа ровно один раз (цикл Эйлера), является легкой, а вот задача построения маршрута, который проходит через каждую вершину графа (гамильтонов цикл) является сложной.

Если вы подозреваете, что задача является NP-полной, сначала проверьте, не содержится ли она в книге [GJ79], в которой приведены несколько сотен известных NP-полных задач. Велика вероятность, что вы найдете свою задачу в этой книге.

Если же нет, то вы можете воспользоваться следующими советами по доказательству сложности задач:

- *Примите меры к тому, чтобы ваша исходная задача была как можно проще (т. е. максимально ограничена).*

  Во-первых, никогда не пытайтесь использовать в качестве исходной задачи общую задачу коммивояжера. Вместо этого используйте задачу поиска гамильтонова цикла, в которой все веса ограничены значением 1 или $\infty$. Еще лучше взять задачу поиска гамильтонова пути, чтобы никогда не нужно было беспокоиться о замыкании цикла. Но самый лучший вариант — это задача поиска гамильтонова пути в ориентированном планарном графе, где степень каждой вершины равна 3. Все эти задачи имеют одинаковую сложность, но чем больше ограничений имеет преобразуемая задача, тем проще окажется процесс сведения.

  Во-вторых, никогда не пытайтесь использовать полную задачу выполнимости для доказательства сложности. Начните с задачи выполнимости 3-SAT. На самом деле,

вам даже не нужно использовать задачу выполнимости полной 3-SAT. Вместо этого можно использовать задачу *планарной 3-SAT*, для которой должен существовать способ представления дизъюнкции в виде плоского графа — такого, что все экземпляры одного и того же литерала можно соединить вместе, избегая пересечения ребер. Указанное свойство полезно при доказательстве неподатливости геометрических задач. Все эти варианты одинаковой сложности, и поэтому сведения NP-полноты с использованием любой из них являются одинаково убедительными.

- *Сделайте целевую задачу максимально сложной.*

  Не бойтесь добавлять дополнительные ограничения или разрешения, чтобы сделать целевую задачу более общей и, следовательно, более сложной. Возможно, задачу на неориентированном графе можно обобщить до задачи на ориентированном графе и, таким образом, только облегчить доказательство ее сложности. Имея доказательство сложности для более сложной задачи, можно возвратиться назад и попытаться упростить целевую задачу.

- *Выбирайте исходную задачу на основании веских аргументов.*

  Выбор правильной исходной задачи играет важную роль в доказательстве сложности. Здесь очень легко ошибиться, хотя теоретически одна NP-полная задача подходит так же хорошо, как и любая другая. Пытаясь доказать сложность задачи, некоторые исследователи просматривают списки с десятками задач в поисках наиболее подходящей. Это непрофессиональный подход. Натолкнувшись на подходящую задачу, они, вероятнее всего, никогда не догадаются, что именно она им и нужна.

  Я использую четыре (и только четыре!) задачи в качестве кандидатов для моих исходных сложных задач. Ограничение количества исходных задач четырьмя означает, что я могу знать многое о каждой из них, — например, какие варианты задачи являются сложными, а какие нет. Приведу свои любимые исходные задачи:

  - *Задача 3-SAT.* Многократно проверенный вариант. Если ни одна из упомянутых далее задач не кажется подходящей, я возвращаюсь к этой первоначальной исходной задаче.
  - *Задача о разделении множества целых чисел.* Это единственно возможный выбор в тех случаях, когда кажется, что для доказательства сложности требуется использовать большие числа.
  - *Задача о вершинном покрытии.* Это подходящий вариант для любой задачи теории графов, чья сложность зависит от *выбора*. Решение задач поиска хроматического числа, клики и независимого множества содержит элемент выбора правильного подмножества вершин или ребер.
  - *Задача о гамильтоновом цикле.* Подходит для любой задачи на графах, чья сложность зависит от *упорядочивания*. Если вы пытаетесь разработать маршрут или календарный план, то задача о гамильтоновом цикле, вероятнее всего, является вашим средством для решения этой задачи.

- *Повышайте стоимость нежелательного выбора.*

  Многие люди испытывают робость при доказательстве сложности задач. Нам нужно преобразовать одну задачу в другую, как можно меньше отступив от оригинальной

задачи. Это легче всего делать, применяя строгие штрафные санкции за любое отклонение от целевого решения. Вы должны мыслить примерно следующим образом: «Если выбирается элемент *a*, то придется выбрать огромное множество *S*, которое не позволит найти оптимальное решение». Чем дороже последствия нежелательных действий, тем легче будет доказать эквивалентность результатов.

- *Разработайте стратегический план, а затем создавайте компоненты для тактических действий.*

  Спрашивайте себя:

  1. Как мне добиться того, чтобы был выбран вариант A или B, но не оба одновременно?
  2. Что сделать, чтобы вариант A был выбран раньше варианта B?
  3. Как поступить с невыбранными вариантами?

  Когда вы разберетесь, какие действия должны выполняться вашими компонентами, то можете задуматься о том, как создать эти компоненты.

- *Если вы испытываете трудности на каком-то этапе, переключайтесь с поиска алгоритма на поиск сведения и обратно.*

  Иногда сложность задачи нельзя доказать по той причине, что для ее решения существует эффективный алгоритм! Применение таких методов, как динамическое программирование или сведение задач к сложным, но полиномиальным задачам на графах — например, к задаче о сопоставлении пар или задаче потоков в сети, может привести к неожиданным результатам. Когда вы не можете доказать сложность задачи, имеет смысл остановиться и попытаться разработать алгоритм, просто для того, чтобы не зацикливаться на одной проблеме.

## 11.7. История из жизни. Наперегонки со временем

Аудитория стремительно теряла внимание к лекции. У некоторых студентов слипались глаза, а другие уже клевали носом.

До конца моей лекции по NP-полноте оставалось двадцать минут, и студентов нельзя было винить за их реакцию. Мы уже рассмотрели несколько сведений, аналогичных представленным в предыдущих разделах. Но сведения NP-полных задач часто легче создать, чем объяснить или понять. Было необходимо продемонстрировать студентам сведение в процессе его создания, чтобы они могли понять, как оно работает.

Я потянулся за книгой «Computers and Intractability» [GJ79], выручавшей меня в трудные моменты. Приложение к ней содержит список из более чем трех сотен разных известных NP-полных задач.

— Итак! — сказал я достаточно громко, чтобы заставить вздрогнуть дремавших в задних рядах. — Доказательства NP-полноты настолько рутинны, что их можно создавать по требованию. Мне нужен помощник. Есть добровольцы?

В передних рядах поднялось несколько рук. Я вызвал к доске одного из студентов.

— Выбери произвольную задачу из списка в приложении в этой книге. Я могу доказать сложность любой из этих задач в течение оставшихся семнадцати минут лекции.

Теперь я определенно овладел их вниманием. Но это было все равно что жонглировать работающими бензопилами. Я должен был выдать результат, не изрезав себя в клочья.

Студент выбрал задачу: «Хорошо, докажите сложность *задачи о неэквивалентности программ с присваиваниями*», — сказал он.

— Никогда раньше не слышал об этой задаче. Прочитай мне условие, чтобы я мог написать его на доске.

Задача определялась таким образом:

> **Задача**. Неэквивалентность программ с присваиваниями (Inequivalence of Programs with Assignments).
>
> **Вход.** Конечное множество переменных $X$, множество их значений $V$ и две программы $P_1$ и $P_2$, каждая из которых состоит из последовательности присваиваний вида:
>
> $$x_0 \leftarrow \text{if } (x_1 = x_2) \text{ then } x_3 \text{ else } x_4,$$
>
> где $x_i$ является элементом $X$.
>
> **Выход.** Возможно ли первоначально присвоить каждой переменной из множества $X$ значение из множества $V$ таким образом, чтобы программы $P_1$ и $P_2$ выдавали разные конечные значения для некой переменной из множества $X$?

Я посмотрел на часы. Пятнадцать минут до конца лекции. Мне предстояло решить языковую задачу. Входом задачи были две программы с переменными, и нужно было проверить, всегда ли они выдают одинаковый результат.

— В первую очередь самое важное. Нам нужно выбрать исходную задачу для нашего сведения. Какая задача подойдет для этого? Задача разделения множества целых чисел? Задача 3-SAT? Задача о вершинном покрытии или о гамильтоновом пути?

Овладев вниманием аудитории, я рассуждал вслух: «Так как наша целевая задача не является задачей на графах или численной задачей, давайте рассмотрим старую добрую задачу 3-SAT. По-видимому, эти две задачи похожи в некоторых аспектах. Задача 3-SAT имеет переменные и эта задача тоже. Чтобы сделать нашу задачу еще более похожей на задачу 3-SAT, можно попробовать ограничить ее переменные только булевыми значения, т. е. $V = \{истина, ложь\}$. Да, это удобно».

До конца лекции оставалось 14 минут.

— В каком направлении должно идти наше сведение? От задачи 3-SAT к программе или от программы к задаче 3-SAT?

Кто-то из первого ряда правильно пробормотал, что от задачи 3-SAT к программе.

— Совершенно верно. Поэтому нам нужно преобразовать наш набор дизъюнкций в две программы. Как мы можем это сделать? Можно рассмотреть вариант разделения дизъюнкций на два набора и написать отдельные программы для каждого из них. Но как их разделить? Я не вижу, как это можно сделать каким-либо естественным способом, поскольку удаление любой дизъюнкции из программы может неожиданно сделать невыполнимую формулу выполнимой, таким образом полностью меняя ответ.

Давайте вместо этого попробуем что-нибудь другое. Мы можем преобразовать все дизъюнкции в одну программу, а потом сделать вторую программу тривиальной. Например, вторая программа может игнорировать вход и всегда выводить или только значение *истина*, или только значение *ложь*. Это выглядит *намного* лучше.

Я продолжал рассуждать вслух. В этом не было ничего особенного. Но я заставил студентов слушать меня!

— Как нам превратить набор дизъюнкций в программу? Мы хотим знать, может ли этот набор дизъюнкций быть выполнен, или, иными словами, существует ли набор значений переменных, делающий его значение истинным. Допустим, что мы создали программу для проверки выполнимости $C_1 = (x_1, \bar{x}_2, x_3)$.

Несколько минут я водил мелом по доске, пока у меня не получилась правильная программа для эмулирования дизъюнкции. Я предположил, что у нас есть доступ к константам для значений *истина* и *ложь*:

$C_1 =$ if ($x_1 =$ *истина*) then *истина* else *ложь*

$C_1 =$ if ($x_2 =$ *ложь*) then *истина* else $C_1$

$C_1 =$ if ($x_2 =$ *истина*) then *истина* else $C_1$

— Теперь у меня есть метод для оценки истинности каждой дизъюнкции. Я могу сделать то же самое в конце для оценки выполнимости всех дизъюнкций.

*sat* = if ($C_1 =$ *истина*) then *истина* else *ложь*

*sat* = if ($C_2 =$ *истина*) then *sat* else *ложь*

...

*sa* = if ($C_n =$ *истина*) then *sat* else *ложь*

В задних рядах возникло оживление. Они увидели луч надежды, что лекция закончится вовремя.

— Прекрасно! У нас имеется программа, которая возвращает истинное значение тогда и только тогда, когда переменным можно присвоить значения так, чтобы выполнялись все дизъюнкции. Нам нужна вторая программа, чтобы закончить доказательство. А если мы напишем *sat* = *ложь*? Да, это всё, что нам нужно. В нашей языковой задаче спрашивается, выводят ли две программы всегда одинаковый результат, независимо от значений, присвоенных переменным. Если дизъюнкции являются выполнимыми, то это означает, что можно присвоить переменным значения таким образом, что длинная программа будет выводить истинное значение. Проверка эквивалентности программ — это то же самое, что и проверка дизъюнкций на выполнимость.

Я победно вскинул руки: «Итак, задача является NP-полной!»

Как только я сказал последнее слово, прозвучал звонок.

## 11.8. История из жизни. Полный провал

Этот педагогический прием с выбором произвольной NP-полной задачи из множества задач, приведенных в книге [GJ79], и доказательством ее сложности на лету мне так понравился, что я начал пользоваться им постоянно. Прием удавался мне восемь раз

подряд. Но так же, как в конце концов завершилась серия из 56 результативных матчей Джо Ди Маджо, и для Google рано или поздно какой-нибудь квартал окажется проигрышным в финансовом отношении, пришло время фиаско и для меня.

На этот раз группа проголосовала за сведение из раздела по теории графов, а студент-доброволец выбрал задачу № 30. Постановка задачи GT30 выглядит таким образом:

**Задача**. Найти односвязный подграф.

**Вход.** Ориентированный граф $G = (V, A)$, положительное целое число $k \leq |A|$.

**Выход.** Существует ли такое подмножество дуг $A' \subseteq A$, где $|A'| \geq k$, для которого в графе $G' = (V, A')$ существует самое большее один ориентированный путь между любой парой вершин?

Мне потребовалось некоторое время, чтобы осмыслить эту задачу. В версии этой задачи для неориентированного графа нужно было бы найти остовное дерево, т. к. оно определяет ровно один путь между любой парой вершин. Добавление в это дерево даже одного ребра $(x, y)$ создало бы цикл — т. е. два разных пути между вершинами $x$ и $y$.

Кроме этого, любая форма ориентированного дерева также была бы односвязной. Но в предложенной задаче нужно найти *наибольший* такой подграф. Рассмотрим двудольный бесконтурный ориентированный граф, состоящий из ориентированных ребер $(l_i, r_j)$, которые все идут из множества «левых» вершин в разные «правые» вершины. Никакой путь в этом графе не содержит более одного ребра, однако граф может содержать $\Omega(n^2)$ ребер.

— Это задача о выборе, — определил я, подумав. — Ведь нам нужно выбрать такое максимально возможное подмножество дуг, которое не содержало бы пары вершин, соединенных множественными путями. А это означает, что предпочтительной задачей для сведения является задача вершинного покрытия.

Я немного поразмыслил о том, чем эти две задачи похожи друг на друга. В обеих задачах нужно найти некоторые подмножества, хотя в задаче о вершинном покрытии требуется найти подмножества вершин, а в задаче об односвязном подграфе — подмножества ребер. Кроме этого, в задаче о вершинном покрытии нужно найти минимально возможное подмножество, в то время как в задаче об односвязном подграфе надо определить максимально возможное подмножество. В исходной задаче имеются неориентированные ребра, а в целевой — ориентированные дуги, так что мне нужно добавить в сведение ориентацию ребер.

Я должен был каким-то образом придать ориентацию ребрам графа вершинного покрытия. Можно было попробовать заменить каждое неориентированное ребро $(x, y)$ дугой, идущей, скажем, от $y$ к $x$. Но в зависимости от выбранного направления дуги получились бы совсем разные ориентированные графы. Поиск «правильной» ориентации ребер мог оказаться трудной задачей, слишком трудной для использования на этапе преобразования.

Я понимал, что ребра можно ориентировать так, чтобы получившийся граф стал бесконтурным ориентированным графом. Но что это дает? В бесконтурных ориентированных графах пары вершин могут быть связаны большим количеством ориентированных путей.

В качестве варианта можно было попробовать заменить каждое неориентированное ребро ($x$, $y$) двумя дугами: от $y$ к $x$ и от $x$ к $y$. Тогда не нужно было бы выбирать правильные дуги для моего сведения, но граф стал бы очень сложным. Я никак не мог найти способ предотвратить множественные нежелательные пути между парами вершин.

Время лекции подходило к концу. В последние десять минут меня охватила паника, т. к. я понял, что на этот раз я не смогу предоставить доказательство.

Нет худшего чувства, чем чувство профессора, завалившего лекцию. Ты стоишь у доски, что-то бормоча, и ясно видишь, что студенты не понимают, о чем идет речь, но догадываются, что ты и сам не понимаешь, о чем говоришь. Прозвучал звонок, и студенты стали покидать аудиторию — одни с сочувствующим выражением на лице, другие с ехидными ухмылками.

Я пообещал им предоставить решение на следующем занятии, но каждый раз, когда я думал об этом, то застревал на одном и том же месте. Я даже попробовал поступить нечестно и найти доказательство в научном журнале, указанном в ссылке в книге [GJ79]. Но ссылка была на неопубликованный доклад 30-летней давности, который нельзя было найти ни в Интернете, ни в библиотеке.

Мысль о следующем занятии, последней лекции семестра, приводила меня в ужас. Но в ночь перед лекцией решение явилось ко мне во сне. «*Раздели каждое ребро пополам*», — сказал мне голос. Я вздрогнул, проснулся и посмотрел на часы. Было три часа ночи.

Вскочив с кровати, я набросал доказательство. Допустим, я заменю каждое неориентированное ребро ($x$, $y$) конструкцией, состоящей из новой центральной вершины $v_{xy}$ с исходящими из нее дугами к вершинам $x$ и $y$ соответственно. Неплохо. Между какими вершинами могут существовать множественные пути? Новые вершины имеют только исходящие ребра, так что только они могут служить источником множественных путей. Старые вершины имеют только входящие ребра. Существует максимум один способ попасть из новой вершины-истока в любую из оригинальных вершин графа вершинного покрытия, так что старые вершины не могли дать множественных путей.

Теперь добавим узел стока $s$ и направим в него ребра из всех оригинальных вершин. От каждой новой вершины к этому стоку будет идти ровно два пути — по одному через каждую из первоначальных вершин, смежных с ней. Один из этих путей нужно разорвать, чтобы создать односвязный подграф. Как это сделать? Для разъединения можно выбрать одну из двух вершин, удалив дугу ($x$, $s$) или ($y$, $s$) новой вершины $v_{xy}$. Чтобы получить подграф максимального размера, мы ищем наименьшее количество дуг, подлежащих удалению. Удалим исходящие дуги хотя бы у одной из двух вершин, определяющих первоначальное ребро. *Но ведь это то же самое, что и поиск вершинного покрытия в этом графе!* Полученное сведение показано на рис. 11.8.

Представление этого доказательства на следующем занятии несколько подняло мою самооценку, т. к. подтвердило правильность провозглашенных мною принципов доказательства сложности задач. Обратите внимание, что в конечном счете сведение не было таким уж сложным — достаточно разорвать ребра и добавить узел стока. Выпол-

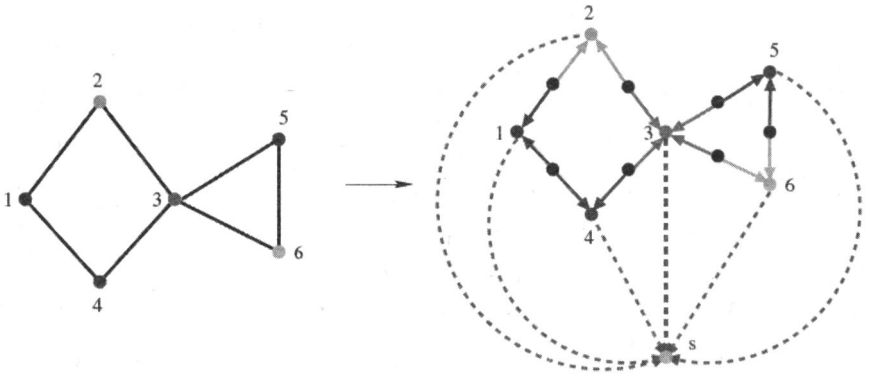



**Рис. 11.8.** Сведение задачи о вершинном покрытии к задаче односвязного подграфа за счет разделения ребер и добавления узла стока

нение сведения NP-полных задач нередко удивляет своей простотой, нужно лишь подойти к нему с правильной стороны.

# 11.9. Сравнение классов сложности P и NP

Теория NP-полноты основана на строгих определениях из теории автоматов и формальных языков. Новички, не обладающие базовыми знаниями, обычно находят применяемую терминологию трудной для понимания или употребляют ее неправильно. Знание детальной терминологии не является чем-то действительно необходимым при решении практических вопросов. Но, несмотря на все сказанное, вопрос: «Верно ли, что P = NP?» представляет собой важнейшую задачу в теории вычислительных систем, и каждый образованный алгоритмист должен понимать, о чем идет речь.

## 11.9.1. Верификация решения и поиск решения

В сравнении классов сложности P и NP главным вопросом является, действительно ли *верификация* решения представляет более легкую задачу, чем первоначальный *поиск* решения. Допустим, что при сдаче экзамена вы «случайно» увидели ответ у студента, сидящего рядом с вами. Принесет ли это вам какую-либо пользу? Вы бы не рискнули сдать этот ответ, не проверив его, поскольку вы способный студент и могли бы решить свою задачу самостоятельно, если бы уделили ей достаточно времени. Но при этом важно, можете ли вы проверить правильность подсмотренного ответа на задачу быстрее, чем решить саму задачу самостоятельно.

В случае NP-полных задач разрешимости, которые обсуждались в этой главе, ситуация *кажется* очевидной. Рассмотрим примеры.

- Можно ли проверить, что предлагаемый для того или иного графа маршрут коммивояжера имеет самое большее вес $k$? Да. Просто сложим вместе веса ребер маршрута и покажем, что общий вес равен самое большее $k$. Это ведь легче, чем найти сам путь с чистого листа, *не так ли*?

- Можно ли проверить, что тот или иной набор значений истинности представляет решение для задачи выполнимости? Да. Просто проверим каждую дизъюнкцию и убедимся, что она содержит, по крайней мере, один истинный литерал из заданного набора значений истинности. Это ведь легче, чем найти выполняющий набор значений истинности с чистого листа, *не так ли*?
- Можно ли проверить, что то или иное подмножество $S$ из $k$ вершин представляет вершинное покрытие графа G? Да. Просто обходим каждое ребро $(u, v)$ графа $G$ и проверяем, что или $u$, или $v$ является элементом $S$. Это ведь легче, чем найти само вершинное покрытие с чистого листа, *не так ли*?

На первый взгляд, все просто. Эти решения можно проверить за линейное время для всех трех задач, тогда как для решения любой из них не известно никакого алгоритма, кроме полного перебора с экспоненциальным временем исполнения. Но проблема в том, что у нас нет строгого *доказательства* нижней границы, препятствующей существованию эффективного алгоритма для решения этих задач. Возможно, в действительности существуют полиномиальные алгоритмы (скажем, с временем исполнения $O(n^{87})$), но мы просто недостаточно хорошо их искали.

## 11.9.2. Классы сложности P и NP

Каждая четко определенная алгоритмическая задача должна иметь максимально быстрый алгоритм решения — точнее говоря, максимально быстрый в терминах «$O$-большое» (Big Oh) для наихудшего случая.

Класс $P$ можно рассматривать как закрытый клуб алгоритмических задач, членом которого могут быть только задачи, для решения которых с чистого листа существует алгоритм с полиномиальным временем исполнения. Полноправными членами этого клуба P являются, например, задача поиска кратчайшего пути, задача поиска минимального остовного дерева и задача календарного планирования. Сокращение P и означает *полиномиальное время исполнения* (Polynomial time).

Менее престижный клуб открывает свои двери любой алгоритмической задаче, решение которой можно *проверить* за полиномиальное время. Как было показано ранее, членами этого клуба являются задача коммивояжера, задача выполнимости и задача о вершинном покрытии — и ни одна из них в настоящее время не квалифицирована должным образом, чтобы стать членом клуба P. Но все члены закрытого клуба P автоматически являются членами этого второго, не столь престижного, клуба. Если задачу разрешимости можно решить с самого начала за полиномиальное время, то ее решение определенно можно проверить столь же быстро: просто решить ее сначала и посмотреть, совпадает ли полученное вами решение с тем, которое было заявлено.

Этот менее престижный клуб называется $NP$, и его название можно расшифровать так: Not necessarily Polynomial time (не обязательно полиномиальное время исполнения)[2].

---

[2] В действительности это сокращение означает non-deterministic polynomial time (недетерминированное полиномиальное время исполнения) и является термином из области теории недетерминированных автоматов.

Важнейший вопрос (как говорят, «вопрос на миллион долларов») заключается в том, содержит ли класс NP задачи, которые не являются членами класса P. Если такой задачи не существует, то классы должны быть одинаковыми и $P = NP$. Но если существует хотя бы одна такая задача, то классы разные и $P \neq NP$. Большинство алгоритмистов и теоретиков сложности вычислений придерживаются мнения, что классы разные, т. е. $P \neq NP$, или, иными словами, что некоторые NP-задачи не могут быть решенными алгоритмом с полиномиальным временем исполнения. Но доказательство этого должно быть гораздо более строгим, чем простое заявление: «Я не могу найти достаточно быстрый алгоритм».

## 11.9.3. Почему задача выполнимости является сложной?

Существует громадное дерево сведений задач NP-полноты, которое всецело основано на сложности задачи выполнимости. Часть задач такого дерева, рассмотренных в этой главе, показана на рис. 11.2.

Но здесь возникает одна тонкость. Каковы были бы последствия, если бы кто-либо *действительно нашел* алгоритм с полиномиальным временем исполнения для решения задачи выполнимости? Существование эффективного алгоритма для любой NP-полной задачи (скажем, задачи коммивояжера) подразумевает существование эффективных алгоритмов для решения всех задач на отрезке пути в дереве сведений между задачей коммивояжера и задачей выполнимости (задача о гамильтоновом цикле, задача о вершинном покрытии и задача 3-SAT). Но существование эффективного алгоритма для рещения задачи выполнимости ничего нам не дает, поскольку путь сведений от задачи SAT к задаче SAT не содержит никаких других задач.

Не будем впадать в панику. Существует замечательное сведение, называемое *теоремой Кука*, которое сводит все задачи класса NP к задаче выполнимости. Таким образом, если доказать, что задача выполнимости, или любая NP-полная задача, является членом класса P, то за ней последуют *все* другие задачи в классе NP, из чего будет следовать, что $P = NP$. Так как, в сущности, каждая упомянутая в этой книге задача является членом класса NP, то результаты такого развития событий были бы весьма значительными и удивительными.

Теорема Кука доказывает, что задача выполнимости является такой же сложной, как и любая другая задача из класса NP. Кроме этого, она также доказывает, что каждая NP-полная задача такая же сложная, как и любая другая. Здесь уместно вспомнить про эффект домино. Но то обстоятельство, что мы не можем найти эффективного алгоритма ни для одной из этих задач, дает веское основание полагать, что все они действительно сложные, а это означает, что $P \neq NP$.

## 11.9.4. NP-сложность по сравнению с NP-полнотой

В завершение имеет смысл обсудить различие между NP-сложностью задачи и ее NP-полнотой. Дело в том, что я имею склонность к несколько вольному обращению с терминологией, а между этими двумя понятиями существует тонкая (обычно несущественная) разница.

Говорят, что задача является *NP-сложной*, если, подобно задаче выполнимости, она, по крайней мере, такая же сложная, как любая другая задача класса NP. Говорят, что задача является *NP-полной*, если она NP-сложная и также является членом класса NP. Поскольку класс NP является большим классом задач, большинство NP-сложных задач, с которыми вам придется столкнуться, в действительности будут членами класса NP и, таким образом, NP-полными. Этот вопрос всегда можно решить, предоставив стратегию проверки решения задачи (обычно достаточно простую). Все рассмотренные в этой книге NP-сложные задачи также являются NP-полными.

Тем не менее существуют некоторые задачи, которые кажутся NP-сложными, но при этом не являются членами класса NP. Такие задачи могут быть *даже более сложными*, чем NP-полные задачи! В качестве примера сложной задачи, не являющейся членом класса NP, можно привести игру для двух участников — такую как шахматы. Представьте себе, что вы сели играть в шахматы с самоуверенным игроком, который играет белыми. Он начинает игру ходом королевской пешки на два поля и объявляет вам мат. Единственным способом доказать его правоту будет создание полного дерева всех ваших возможных ходов и его лучших ответных ходов и демонстрация невозможности вашего выигрыша в текущей позиции. Количество узлов этого полного дерева будет экспоненциально зависеть от его высоты, равной количеству ходов, которые вы сможете сделать, перед тем как проиграть, применяя максимально эффективную защиту. Ясно, что это дерево нельзя создать и проанализировать за полиномиальное время, поэтому задача не является членом класса NP.

## Замечания к главе

Понятие NP-полноты было впервые сформулировано Стивеном Куком (Stephen Cook) в 1971 г. (см. [Coo71]). Задача выполнимости действительно является задачей «на миллион долларов», поскольку институт Clay Mathematical Institute предлагает премию такого размера любому, кто решит вопрос $P = NP$. Подробности о задаче и призе за ее решение см. по адресу **http://www.claymath.org**.

В 1972 г. Ричард Карп (Richard Karp) продемонстрировал значимость работы Кука, предоставив сведение от задачи выполнимости к каждой из более чем 20 важных алгоритмических задач. Я рекомендую вам ознакомиться с докладом Карпа ([Kar72]) в силу его исключительного изящества и краткости — он излагает каждое сведение в трех строчках описания, показывающего эквивалентность задач. Работы Кука и Карпа, вместе взятые, предоставляют инструменты для разрешения вопроса сложности буквально сотен важных задач, для решения которых не было известно эффективных алгоритмов.

Наилучшим введением в теорию NP-полноты остается книга «Computers and Intractability» ([GJ79]). В ней дается введение в общую теорию, включающее доступное доказательство теоремы Кука ([Coo71]) о том, что сложность задачи выполнимости такая же, как и любой другой задачи класса NP. Книга также содержит важный справочный каталог свыше 300 NP-полных задач, что является хорошим материалом для изучения известных фактов о наиболее интересных задачах. Задачи сведения, упомянутые, но не рассмотренные в этой главе, можно найти в книгах [GJ79] и [CLRS09].

В захватывающем романе «Factor Man» [Gin18] герой открывает полиномиальный алгоритм для решения задачи выполнимости, в результате чего его начинают преследовать государственные агенты и наемные убийцы. Я победно поднимаю за эту книгу два пальца вверх. В книге «Golden Ticket» [For13] доступно излагается теория сложности и вопрос $P = NP$.

Несколько задач из каталога находятся в состоянии неопределенности, т. е. неизвестно, существует ли эффективный алгоритм для решения конкретной задачи или же она является NP-полной. Наиболее значительными из этих задач являются задачи изоморфизма графов (см. *разд. 19.9*) и разложения целых чисел на множители (см. *разд. 16.8*). Краткость этого списка задач с неопределенным статусом в большой степени является следствием современного уровня развития дисциплины разработки алгоритмов и мощи теории NP-полноты. Почти для каждой важной задачи у нас имеется либо эффективный алгоритм, либо веская причина его отсутствия.

Альтернативный и вдохновляющий взгляд на предмет NP-полноты предоставляется в видеоуроках Эрика Демейна (Erik Demain) для курса Массачусетского технологического института «Algorithmic Lower Bounds: Fun with Hardness Proofs» (Нижние границы алгоритмов: развлекаемся с доказательствами сложности) (**http://courses.csail.mit.edu/6.890/fall14/**).

Задача об односвязном подграфе, которая рассматривалась в *разд. 9.8*, впервые была доказана в техническом докладе [Mah76].

# 11.10. Упражнения

## Преобразования и выполнимость

1. *[2]* Предоставьте формулу 3-SAT, которая получается в результате применения сведения задачи выполнимости к задаче 3-SAT для следующей формулы:

$$(x \vee y \vee \bar{z} \vee w \vee u \vee \bar{v}) \wedge (\bar{x} \vee \bar{y} \vee z \vee \bar{w} \vee u \vee v) \wedge (x \vee \bar{y} \vee \bar{z} \vee w \vee u \vee \bar{v}) \wedge (x \vee \bar{y})$$

2. *[3]* Нарисуйте граф, который получается в результате сведения задачи 3-SAT к задаче о вершинном покрытии для выражения

$$(x \vee \bar{y} \vee z) \wedge (\bar{x} \vee y \vee \bar{z}) \wedge (\bar{x} \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{x}).$$

3. *[3]* Докажите, что задача 4-SAT является NP-сложной.

4. *[3] Экономная задача SAT* (stingy SAT) ставится следующим образом. Для заданного множества дизъюнкций литералов и целочисленного значения $k$ нужно найти такой выполняющий набор значений истинности, в котором значение *истина* имеют самое большее $k$ переменных, если такой набор существует. Докажите, что экономная задача SAT является NP-сложной.

5. *[3]* В задаче *двойного SAT* (double SAT) требуется определить, имеет ли та или иная задача выполнимости, по крайней мере, два разных выполняющих набора значений истинности. Например, задача $\{\{v_1, v_2\}, \{\bar{v}_1, v_2\}, \{\bar{v}_1, \bar{v}_2\}\}$ выполнима, но имеет только одно решение ($v_1 = F$, $v_2 = T$). Противоположно ей задача $\{\{v_1, v_2\}, \{\bar{v}_1, \bar{v}_2\}\}$ имеет ровно два решения. Докажите, что задача двойного SAT является NP-сложной.

6. *[4]* Допустим, что у нас имеется процедура, которая может определить разрешимость задачи коммивояжера из *разд. 11.1.2* за, скажем, линейное время. Предоставьте эффективный алгоритм поиска самого маршрута, вызывая эту процедуру полиномиальное число раз.

7. *[7]* Реализуйте процедуру сведения SAT в SAT-3 для преобразования экземпляров задач выполнимости в эквивалентные экземпляры задачи 3-SAT.

8. *[7]* Разработайте и реализуйте алгоритм поиска с возвратом для проверки одного множества из множества дизъюнкций на выполнимость. Какие критерии можно использовать для отсечения пространства поиска?

9. *[8]* Реализуйте процедуру сведения задачи о вершинном покрытии к задаче выполнимости и проверьте получившиеся дизъюнкции программой проверки на выполнимость. Насколько такой способ применим на практике для выполнения подобных вычислений?

## Базовые сведения

10. *[4]* Дано: множество $X$ из $n$ элементов, семейство $F$ подмножеств множества $X$ и целое число $k$. Существует ли $k$ подмножеств семейства $F$, объединение которых равно $X$?

    Например, если $X = \{1,2,3,4\}$ и $F = \{\{1,2\},\{2,3\},\{4\},\{2,4\}\}$, для $k = 2$ решения нет, но для $k = 3$ есть (например, {1,2}, {2,3}, {4}).

    Докажите, что задача о покрытии множества является NP-сложной, выполнив сведение с задачей вершинного покрытия.

11. *[4]* Задача *коллекционера бейсбольных карточек* выглядит таким образом. Имеются пакеты бейсбольных карточек $P_1$, ..., $P_m$, причем каждый из них содержит подмножество карточек, выпущенных в указанном году. Возможно ли собрать все карточки за этот год, купив не более чем $k$ таких пакетов?

    Например, для игроков {*Aaron*, *Mays*, *Ruth*, *Skiena*} и пакетов:

    $$\{\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}, \{Mays, Skiena\}\}$$

    для $k = 2$ решения нет, но для $k = 3$ есть:

    $$\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}$$

    Докажите, что задача коллекционера бейсбольных карточек является NP-полной, используя для этого сведение к ней от задачи о вершинном покрытии.

12. *[4]* Рассмотрим *задачу остовного дерева низкой степени* (low-degree spanning tree problem). Дано: граф $G$ и целое число $k$. Содержит ли граф $G$ такое остовное дерево, степень всех вершин которого равна самое большее $k$? (Очевидно, что степень определяется только по количеству ребер.) Например, граф, представленный на рис. 11.9, не содержит остовного дерева, степень всех вершин которого равна самое большее трем.

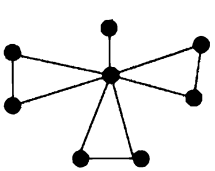


**Рис. 11.9.** Пример графа

a) Докажите, что задача остовного дерева низкой степени является NP-полной, используя сведение от задачи о гамильтоновом *пути*.

b) Теперь сформулируем *задачу остовного дерева высокой степени* (high-degree spanning tree problem). Дано: граф $G$ и целое число $k$. Содержит ли граф $G$ остовное дерево, у которого максимальная степень вершины равна как минимум $k$? В предыдущем примере обсуждалось остовное дерево с наивысшей степенью, равной 7. Предоставьте эффективный алгоритм для решения задачи остовного дерева высокой степени и выполните анализ его временной сложности.

13. *[5]* В задаче о *покрытии множества с минимальным элементом* (minimum element set cover) требуется найти такое покрытие $S \subseteq C$ универсального множества $U = \{1, ..., n\}$, для которого сумма размеров подмножеств в $S$ была бы самое большее $k$.

    a) Покажите, что для $C = \{\{1, 2, 3\}, \{1, 3, 4\}, \{2, 3, 4\}, \{3, 4, 5\}\}$ существует покрытие размером 6, но ни одного покрытия размером 5 из-за повторяющегося элемента.

    b) Докажите, что эта задача является NP-сложной. (Подсказка: задача покрытия множества является сложной, если все подмножества имеют одинаковый размер.)

14. *[3]* В задаче *половинного гамильтонова цикла* (half-Hamiltonian cycle) для заданного графа $G$ с $n$ вершинами требуется определить, содержит ли этот граф простой цикл длиной ровно $\lfloor n/2 \rfloor$, где функция `floor` округляет входной аргумент до ближайшего целого в меньшую сторону. Докажите, что эта задача является NP-сложной.

15. *[5]* В задаче *равномерного распределения нагрузки по трем фазам* требуется разделить множество из $n$ натуральных чисел на три множества: $A$, $B$ и $C$ таким образом, чтобы $\sum_i a_i = \sum_i b_i = \sum_i c_i$. Докажите, что эта задача является NP-сложной, используя для этого сведение от задачи разделения множества целых чисел или задачи суммы подмножества (см. *разд. 10.5*).

16. *[4]* Докажите, что следующая задача является NP-сложной:

    **Задача**. Найти плотный подграф.

    **Вход.** Граф $G$ и два целых числа $k$ и $y$.

    **Выход.** Содержит ли граф $G$ подграф, имеющий ровно $k$ вершин и, по крайней мере, $y$ ребер?

17. *[4]* Докажите, что следующая задача является NP-сложной:

    **Задача**. Найти клику.

    **Вход.** Неориентированный граф $G = (V, E)$ и целое число $k$.

    **Выход.** Содержит ли граф $G$ как клику, так и независимое множество размером $k$?

18. *[5] Эйлеровым циклом* (Eulerian cycle) называется маршрут, который проходит по каждому ребру графа ровно один раз. *Эйлеровым подграфом* (Eulerian subgraph) называется подмножество ребер и вершин графа, которое содержит эйлеров цикл. Докажите, что задача определения количества ребер в наибольшем эйлеровом подграфе графа является NP-сложной. (Подсказка: задача о гамильтоновом цикле является NP-сложной, даже если каждой вершине графа инцидентны три ребра.)

19. *[5]* Покажите, что следующая задача является NP-сложной:

    **Задача**. Максимальный общий подграф.

    **Ввод**. Два графа $G_1 = (V_1, E_1)$ и $G_2 = (V2, E_2)$ и бюджет $b$.

**Вывод.** Два множества узлов: $S_1 \subseteq V_1$ и $S_2 \subseteq V_2$, удаление которых оставит в каждом графе, по крайней мере, $b$ узлов и сделает оба графа идентичными.

20. *[5] Сильно независимое множество* (strongly independent set) — это такое подмножество вершин $S$ в графе $G$, между любыми двумя вершинами которого нет пути длиной 2. Докажите, что эта задача является NP-сложной.

21. *[5] Воздушным змеем* (kite) называется граф с четным количеством вершин, скажем, $2n$, в котором $n$ вершин образуют клику, а остальные $n$ вершин соединены в хвост, который состоит из пути, соединенного с одной из вершин клики. В задаче *максимального графа воздушного змея* для заданного графа и цели $g$ требуется найти такой подграф, который является графом воздушного змея и содержит $2g$ узлов. Докажите, что эта задача является NP-сложной.

## Нестандартные сведения

22. *[5]* Докажите, что следующая задача является NP-сложной:

    **Задача**. Найти минимальное множество представителей (hitting set).

    **Вход.** Коллекция $C$ подмножеств множества $S$, положительное число $k$.

    **Выход.** Содержит ли множество $S$ такое подмножество $S'$, для которого $|S'| \leq k$ и каждое подмножество в коллекции $C$ содержит, по крайней мере, один элемент из подмножества $S'$?

23. *[5]* Докажите, что следующая задача является NP-сложной:

    **Задача**. Задача о рюкзаке.

    **Вход.** Множество $S$, состоящее из $n$ элементов, в котором $i$-й элемент имеет ценность $v_i$ и вес $w_i$. Два положительных целых числа: максимальный вес $W$ и требование к ценности груза $V$.

    **Выход.** Существует ли такое подмножество $S' \subseteq S$, для которого $\sum_{i \in S'} w_i \leq W$ и $\sum_{i \in S'} v_i \geq V$ ? (Подсказка: сначала рассмотрите задачу разделения множества целых чисел.)

24. *[5]* Докажите, что следующая задача является NP-сложной:

    **Задача.** Найти гамильтонов путь.

    **Вход.** Граф $G$ и вершины $s$ и $t$.

    **Выход.** Содержит ли граф $G$ путь с началом в вершине $s$ и концом в вершине $t$, который проходит через все вершины не более одного раза? (Подсказка: начните с задачи о гамильтоновом цикле.)

25. *[5]* Докажите, что следующая задача является NP-сложной:

    **Задача.** Найти самый длинный путь.

    **Вход.** Граф $G$ и положительное целое число $k$.

    **Выход.** Содержит ли граф $G$ путь, который соединяет, по крайней мере, $k$ разных вершин, не проходя через каждую из них более одного раза?

26. *[6]* Докажите, что следующая задача является NP-сложной:

    **Задача.** Найти доминирующее множество.

    **Вход.** Граф $G = (V, E)$ и положительное целое число $k$.

    **Выход.** Существует ли такое подмножество $V' \subseteq V$, где $|V'| \leq k$, в котором для каждой вершины $x \in V$ или $x \in V'$, или существует ребро $(x, y) \in E$, где $y \in V'$.

27. *[7]* Докажите, что задача о вершинном покрытии (существует ли такое подмножество $S$, состоящее из $k$ вершин графа $G$, в котором каждое ребро в графе $G$ инцидентно, по крайней мере, одной вершине в $S$?) остается NP-сложной, даже если степень всех вершин в графе может быть только четной?

28. *[7]* Докажите, что следующая задача является NP-сложной:

    **Задача.** Задача упаковки множества (set packing problem).

    **Вход.** Коллекция $C$ подмножеств множества $S$, положительное число $k$.

    **Выход.** Содержит ли $C$, по крайней мере, $k$ непересекающихся подмножеств, т. е. подмножеств, ни одна пара из которых не имеет общих элементов?

29. *[7]* Докажите, что следующая задача является NP-сложной:

    **Задача.** Найти разрывающее множество вершин.

    **Вход.** Ориентированный граф $G = (V, A)$ и положительное целое число $k$.

    **Выход.** Существует ли такое подмножество $V' \subseteq V$, где $|V'| \leq k$, удаление вершин которого из графа $G$ оставляет бесконтурный ориентированный граф?

30. *[8]* Предоставьте сведение от задачи судоку к задаче вершинной раскраски графа. В частности, опишите, как для частично заполненной доски судоку создать граф, который можно раскрасить девятью цветами тогда, и только тогда, когда для этой доски судоку существует решение.

## Алгоритмы для решения частных случаев задач

31. *[5]* Гамильтонов путь $P$ проходит через каждую вершину ровно один раз. Задача поиска гамильтонова пути в графе $G$ является NP-полной. В отличие от гамильтонова цикла, гамильтонов путь не обязан содержать ребро, соединяющее начальную и конечную вершины пути $P$.

    Предоставьте алгоритм с временем исполнения $O(n + m)$ для проверки бесконтурного ориентированного графа $G$ на наличие в нем гамильтонова пути. (Подсказка: ваши рассуждения должны идти в направлении топологической сортировки и обхода в глубину.)

32. *[3]* Задача $k$-клики — это общая задача клики, ограниченная графами, степень любой вершины которых не больше чем $k$. Докажите, что существует алгоритм для решения задачи $k$-клики для любого заданного значения $k$ — т. е. что $k$ является константой.

33. *[8]* В задаче 2-SAT требуется выяснить, является ли выполнимой заданная булева формула в 2-конъюктивной нормальной форме (КНФ). Задача 2-SAT подобна задаче 3-SAT, только каждая дизъюнкция может содержать всего лишь два литерала. Например, следующая формула записана в 2-КНФ:

$$(x_1 \vee x_2) \wedge (\overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_3).$$

Предоставьте алгоритм с полиномиальным временем исполнения для решения задачи 2-SAT.

## P или NP?

34. *[4]* Покажите, что следующие задачи принадлежат классу NP:
    - Содержит ли граф $G$ простой путь (т. е. путь, в котором вершины не повторяются) длиной $k$?
    - Является ли целое число $n$ составным (т. е. не простым)?
    - Содержит ли граф $G$ вершинное покрытие размером $k$?
35. *[7]* До 2002 года оставался открытым вопрос, возможно ли за полиномиальное время найти решение такой задачи разрешимости: «Является ли целое число $n$ составным, т. е. не простым?» Почему следующий алгоритм не доказывает, что эта задача принадлежит классу P, хотя он и исполняется за время $O(n)$?

```
PrimalityTesting(n)
   composite = false
   for i := 2 to n - 1 do
      if (n mod i) = 0 then
         composite = true
```

## LeetCode

1. **https://leetcode.com/problems/target-sum/**
2. **https://leetcode.com/problems/word-break-ii/**
3. **https://leetcode.com/problems/number-of-squareful-arrays/**

## HackerRank

1. **https://www.hackerrank.com/challenges/spies-revised**
2. **https://www.hackerrank.com/challenges/brick-tiling/**
3. **https://www.hackerrank.com/challenges/tbsp/**

## Задачи по программированию

Эти задачи доступны на сайте **http://onlinejudge.org**:

1. «The Monocycle», глава 12, задача 10047.
2. «Dog and Gopher», глава 12, задача 111301.
3. «Chocolate Chip Cookies», глава 12, задача 10136.
4. «Birthday Cake», глава 12, задача 10167.

> ***Примечание***
>
> Эти задания не имеют непосредственного отношения к теме NP-полноты, но добавлены для полноты картины.

# Решение сложных задач

Человек практичный никогда не останавливается на доказательстве того, что задача является NP-полной. Очевидно, у него есть какие-то причины решить ее. Эти причины не исчезли, когда он узнал, что для решения задачи не существует алгоритма с полиномиальным временем исполнения. Ему все равно нужна программа для решения этой задачи. Все, что известно, так это лишь невозможность создания программы для быстрого оптимального решения задачи в наихудшем случае.

Однако для достижения цели остаются три варианта:

- *алгоритмы, эффективные для средних случаев задачи.*

  В качестве примеров таких алгоритмов можно назвать алгоритмы поиска с возвратом, в которых выполняются значительные отсечения.

- *эвристические алгоритмы.*

  Эвристические методы — такие как имитация отжига или «жадные» алгоритмы — можно использовать, чтобы быстро найти решение, но без гарантии, что это решение будет наилучшим.

- *аппроксимирующие алгоритмы.*

  Теория NP-полноты только оговаривает сложность получения *точного* решения задачи. Но, используя специализированные, ориентированные на конкретную задачу эвристические алгоритмы, можно получить доказуемо *близкое* к оптимальному решение для всех возможных экземпляров задачи.

В этой главе мы исследуем указанные возможности более глубоко. Кроме того, мы вкратце рассмотрим предмет квантовых вычислений. Эта захватывающая технология сотрясает (но не разрывает) границы эффективного вычисления задач.

## 12.1. Аппроксимирующие алгоритмы

Аппроксимирующие алгоритмы обеспечивают решение с гарантией, в частности, того, что качество оптимального решения доказуемо ограничено качеством эвристического алгоритма. То есть, независимо от входного экземпляра и уровня вашего везения, такой аппроксимирующий алгоритм неизбежно выдаст правильный ответ. Кроме этого, доказуемо качественные аппроксимирующие алгоритмы часто концептуально просты, работают быстро и легко поддаются программированию.

Однако неясным остается следующее обстоятельство. Насколько решение, полученное с помощью аппроксимирующего алгоритма, сравнимо с решением, которое можно было бы получить с помощью эвристического алгоритма, не дающего никаких гарантий? Вообще говоря, оно может оказаться как лучше, так и хуже. Положив деньги в банк, вы

получаете гарантию невысоких процентов прибыли безо всякого риска. Вложив эти же деньги в акции, вы, скорее всего, получите более высокую прибыль, но при этом у вас не будет никаких гарантий.

Один из способов получения наилучшего результата от аппроксимирующего и простого эвристического алгоритмов заключается в том, что вы решаете свой экземпляр задачи с помощью каждого из них и выбираете тот алгоритм, который дает лучшее решение. При этом вы получаете гарантированное решение *и* дополнительный шанс на получение лучшего решения. Применяя эвристические алгоритмы для решения сложных задач, вы можете рассчитывать на двойной успех.

## 12.2. Аппроксимация вершинного покрытия

Вспомним задачу вершинного покрытия, в которой требуется найти такое небольшое подмножество $S$ вершин графа $G$, что хотя бы одна вершина каждого ребра $(x, y)$ является членом $S$. Как мы уже видели, задача поиска минимального вершинного покрытия графа является NP-полной. Но с помощью очень простой процедуры (листинг 12.1) можно всегда найти покрытие, которое самое большее вдвое превышает оптимальное. Эта процедура многократно выбирает ребро с не входящими в покрытие вершинами и добавляет *обе* эти вершины в покрытие.

**Листинг 12.1. Аппроксимирующий алгоритм поиска вершинного покрытия графа**

```
VertexCover(G = (V, E))
   While (E ≠ 0) do:
      Выбираем произвольное ребро (u, v) ∈ E
      Добавляем обе вершины u и v к вершинному покрытию
      Удаляем из E все ребра, входящие в u или v
```

Должно быть очевидным, что такая процедура всегда выдает вершинное покрытие, поскольку каждое ребро удаляется только после добавления к покрытию инцидентной вершины. Более интересным является утверждение, что наилучшее вершинное покрытие должно содержать, по крайней мере, в два раза меньше вершин этого покрытия. Почему? Рассмотрим только те $k$ ребер, выбранные алгоритмом, которые образуют в графе паросочетание. Ни одна пара таких ребер в паросочетаниях не может иметь общую вершину. Поэтому *любое* покрытие, состоящее только из этих $k$ ребер, должно включать хотя бы одну вершину на каждое ребро, что делает его, по крайней мере, вдвое меньшим, чем вершинное покрытие с $2k$ вершинами, полученное с помощью этого «жадного» алгоритма.

Стоит отметить несколько интересных аспектов этого алгоритма.

- *Хотя процедура проста, она не так глупа.*

  Производительность многих кажущихся более интеллектуальными эвристических алгоритмов может оказаться намного ниже в наихудшем случае. Например, почему бы не модифицировать эту процедуру, чтобы для получения вершинного покрытия вместо обеих вершин выбирать только одну из них? В конце концов, выбранное ребро будет с тем же успехом покрыто только одной вершиной. Но посмотрим на

звездообразный граф, показанный на рис. 12.1. Если не выбрать центральную вершину, вершинное покрытие окажется крайне неудачным.

Первоначальный эвристический алгоритм выдаст двухвершинное покрытие, в то время как эвристический алгоритм, выбирающий одну вершину, *может* выдать покрытие размером до $n - 1$ вершин, и, если нам не повезет, алгоритм будет постоянно выбирать листовой узел вместо центрального в качестве вершины покрытия, которую следует сохранить.

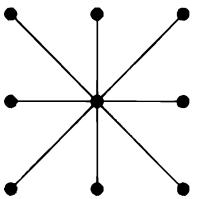


**Рис. 12.1.** Звездообразный граф

- *«Жадный» алгоритм — это не всегда то, что нужно.*

Возможно, самый естественный алгоритм поиска вершинного покрытия будет постоянно выбирать (а затем удалять) вершину наивысшей оставшейся степени для этого вершинного покрытия. В конце концов, эта вершина покроет наибольшее количество возможных ребер. Но в случае необходимости выбора между одинаковыми или почти одинаковыми вершинами этот эвристический алгоритм может значительно отклониться от правильного пути. В наихудшем варианте он может выдать покрытие больше оптимального в $\Theta(\lg n)$ раз, как показано в примере, приведенном на рис. 12.2.

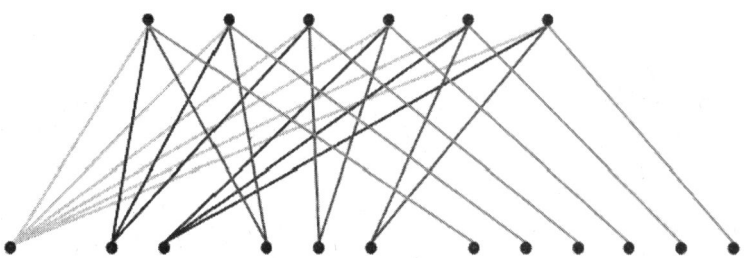


**Рис. 12.2.** Пример нахождения эвристическим алгоритмом плохого вершинного покрытия

Оптимальным покрытием для такого двудольного графа будет верхний ряд вершин, но «жадный» алгоритм выбирает слева направо вершины нижнего ряда. Этот пример можно расширить, чтобы создать экземпляр задачи, для которой решение «жадного» алгоритма будет в $\Theta(\log n)$ раз больше, чем минимальное вершинное покрытие.

- *Усложнение эвристического алгоритма не обязательно улучшает его.*

Эвристический алгоритм легко усложнить, вставляя в него дополнительные возможности обработки. Например, в предыдущей аппроксимирующей процедуре не

указывается, какое ребро выбирается следующим. Может показаться разумным следующим выбирать ребро с вершинами наивысшей общей степени. Но это не сузит границы наихудшего случая, а только сделает более трудным анализ работы алгоритма.

- *Корректное завершение алгоритма позволяет улучшить результат.*

  Дополнительным преимуществом создания простых эвристических алгоритмов является то, что их часто можно модифицировать для получения лучших практических решений, не ослабляя при этом пределы аппроксимации. Например, операция постобработки, которая удаляет из покрытия все ненужные вершины, на практике может только улучшить результат, даже если не принесет никакой пользы теоретическому пределу наихудшего случая. Также можно многократно повторить этот процесс с разными исходными ребрами и выбрать наилучший из полученных результатов.

Важной характеристикой аппроксимирующих алгоритмов является отношение размера полученного решения к нижней границе оптимального решения. Не следует размышлять о том, насколько хорошим могло бы быть наше решение, — мы должны думать о наихудшем случае, т. е. о том, насколько плохим оно могло бы быть.

### ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Вершинное покрытие в остатке

**ЗАДАЧА**. Выполняем поиск обходом в глубину графа $G$, создавая в процессе дерево глубинного поиска $T$. Конечными узлами (листьями) в этом дереве являются любые некорневые узлы со степенью 1. Удаляем из дерева $T$ все концевые узлы. Докажите, что:

1. Множество всех неконцевых узлов дерева $T$ составляет вершинное покрытие графа $G$.
2. Размер этого вершинного покрытия не больше чем вдвое превышает размер минимального вершинного покрытия.

**РЕШЕНИЕ**. Почему множество всех нелистовых узлов дерева обхода в глубину $T$ должно создавать вершинное покрытие? Вспомним волшебное свойство обхода в глубину — он разделяет все ребра дерева на древесные и обратные. Если вершина $v$ является концевым узлом дерева $T$, тогда существует единственное древесное ребро $(x, v)$, содержащее эту вершину, которая останется в покрытии в результате удаления нелистовой вершины $x$. Если вершина $v$ содержится в других ребрах, эти ребра должны быть обратными, идущими к предшественникам вершины $v$, которые все были выбраны для включения в покрытие. Поэтому все ребра будут охвачены множеством нелистовых узлов.

Но почему размер множества нелистовых узлов не более чем вдвое превышает оптимальное покрытие? Выполним обход дерева, начиная с любого листового узла $v$ и двигаясь по направлению к корню. Предположим, что длина такого пути составляет $k$ ребер, а это означает, что полный путь от листа к корню содержит $k + 1$ вершину. Наш эвристический алгоритм выберет для вершинного покрытия эти $k$ нелистовые вершины. Но для наилучшего вершинного покрытия для этого пути требуется k/2 вершин, поэтому размер получаемого таким образом пути всегда не больше чем вдвое превышает размер оптимального покрытия. ■

## 12.2.1. Рандомизированный эвристический алгоритм вершинного покрытия

Ранее мы доказали, что наш первоначальный эвристический алгоритм для вычисления вершинного покрытия посредством выбора произвольного ребра, вершины которого не входят в покрытие, и добавлением обеих его вершин в покрытие является двукратным аппроксимирующим алгоритмом. Но расширение покрытия двумя вершинами, когда только одной из них было бы достаточно для этого ребра, не кажется правильным. Кроме того, как можно видеть на примере звездообразного графа, показанного на рис. 12.1, если для каждого ребра постоянно выбирать неправильную (т. е. нецентральную) вершину, размер полученного вершинного покрытия может оказаться размером $n - 1$ вершин вместо 1.

Для столь ужасной эффективности требуется принять $n - 1$ неправильных решений подряд, что подразумевает или особый талант, или чрезвычайное невезение. Сделать эффективность алгоритма зависимой от везения можно, выбирая вершины наугад, как показано в листинге 12.2.

**Листинг 12.2. Алгоритм вершинного покрытия с произвольным выбором вершин**

```
VertexCover(G = (V,E))
While (E _≠ 0) do:
    Выбираем произвольное ребро (u, v) ∈ E
    Выбираем вершину u и v ребра и добавляем их к вершинному покрытию
    Удаляем из E все ребра, входящие в выбранную вершину
```

По завершении исполнения этой процедуры мы получим вершинное покрытие, но насколько хорош его ожидаемый размер по сравнению с размером определенного минимального покрытия $C$? Обратите внимание на то, что с каждым выбираемым ребром $(u, v)$ в оптимальное покрытие $C$ должен добавляться, по крайней мере, один из двух концевых узлов. То есть нам везет, и мы выбираем «правильную» вершину как минимум в половине случаев. По завершении этой процедуры у нас будет выбрано множество $C' \subset C$ вершин покрытия плюс множество вершин $D$ из $V - C$ для нашего покрытия. Мы знаем, что $|C'|$ всегда должно быть меньшим или равным $|C|$. Кроме этого, ожидаемый размер множества $D$ равен размеру $C'$. Таким образом, ожидается: $|C'| + |D| \leq 2|C|$, и мы получаем решение с ожидаемым размером, не больше чем вдвое превышающим размер оптимального решения.

Рандомизация — очень мощное средство для разработки аппроксимирующих алгоритмов. Ее роль заключается в том, чтобы устранить неблагоприятные особые случаи, делая их возникновение маловероятным. Для тщательного анализа таких вероятностей часто требуются тонкие методы, но сами эвристические алгоритмы обычно очень простые и легко поддаются реализации.

# 12.3. Задача коммивояжера в евклидовом пространстве

В большинстве естественных приложений задачи коммивояжера прямые маршруты по своей природе короче, чем обходные. Например, если в качестве веса ребер графа выбрать расстояние между городами по прямой, то кратчайший путь от $x$ к $y$ всегда будет проходить по прямой.

Веса ребер, назначаемые в соответствии с евклидовой геометрией, удовлетворяют *аксиоме треугольника*, гласящей, что для любой тройки вершин $u$, $v$ и $w$ выполняется неравенство $d(u, w) \leq d(u, v) + d(v, w)$. Интуитивная очевидность этого условия показана на рис. 12.3.

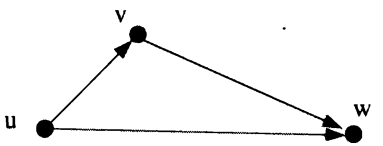



**Рис. 12.3.** Аксиома треугольника: $d(u, w) \leq d(u, v) + d(v, w)$ — применима для расстояний между точками геометрических фигур

Вместе с тем стоимость билета на авиарейс представляет собой пример функции расстояний, которая нарушает аксиому треугольника, поскольку иногда транзитный рейс обходится дешевле, чем прямой полет до города назначения. Этим и объясняется, почему подбор самого дешевого авиабилета может быть сопряжен с большими трудностями. Но аксиома треугольника тем не менее естественно применима для многих задач и приложений.

Задача коммивояжера остается сложной, когда веса ребер определяются евклидовыми расстояниями между точками. Но оптимальный маршрут коммивояжера на графах, подчиняющихся аксиоме треугольника, можно аппроксимировать, используя минимальные остовные деревья. Прежде всего, обратите внимание на то, что вес минимального остовного дерева графа $G$ должен быть нижней границей стоимости оптимального маршрута коммивояжера $T$ в графе $G$. Почему? Расстояния всегда положительные, поэтому удаление любого ребра из маршрута $T$ оставляет путь, общий вес которого должен быть не большим, чем вес первоначального маршрута. Этот путь не содержит циклов, вследствие чего он является деревом, а это означает, что его вес должен быть равен, *по крайней мере*, весу минимального остовного дерева. Таким образом, вес минимального остовного дерева дает нам нижнюю границу стоимости оптимального маршрута коммивояжера.

Теперь рассмотрим, что происходит при обходе остовного дерева в глубину. Каждое ребро мы посещаем дважды: один раз спускаясь по дереву при открытии ребра, а второй — возвращаясь наверх после исследования всего поддерева. Например, при таком обходе в глубину графа, показанного на рис. 12.4, вершины посещаются в следующем порядке:

1, 2, 1, 3, 5, 8, 5, 9, 5, 3, 6, 3, 1, 4, 7, 10, 7, 11, 7, 4, 1.

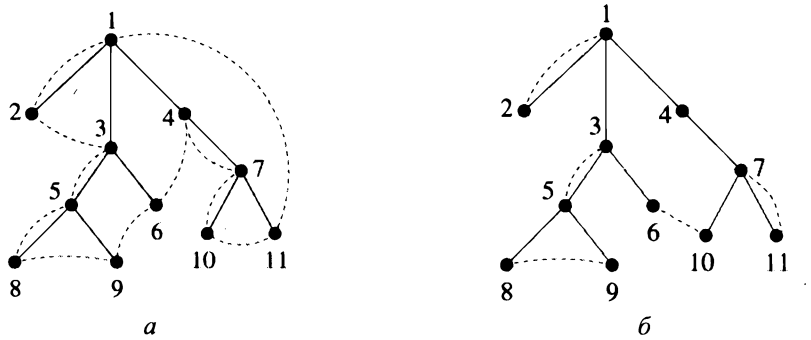



**Рис. 12.4.** Обход в глубину остовного дерева по укороченному маршруту (*а*). То же самое дерево обхода в глубину при сравнении минимальных весов вершин нечетных степеней, создающем эйлеров граф для эвристического алгоритма Кристофидеса (*б*)

Этот маршрут проходит по каждому ребру минимального остовного дерева дважды и, следовательно, стоит самое меньшее вдвое дороже, чем оптимальный маршрут.

Но многие вершины на этом маршруте обхода в глубину будут повторяться. Поэтому, чтобы удалить лишние вершины, на каждом шаге можно выбирать прямой путь к следующей непосещенной вершине. Тогда кратчайший маршрут для нашего дерева будет проходить через следующие вершины (рис. 12.4, *а*):

1, 2, 3, 5, 8, 9, 6, 4, 7, 10, 11, 1.

Так как мы заменили последовательность ребер одним направленным ребром, аксиома треугольника обеспечивает ситуацию, при которой маршрут может стать только короче. В результате вес этого укороченного маршрута, который для графа $G$ с $n$ вершинами и $m$ ребрами можно составить за время $O(n + m)$, всегда будет не более чем в два раза превышать вес оптимального маршрута коммивояжера на графе $G$.

## 12.3.1. Эвристический алгоритм Кристофидеса

На эту идею — удваивания минимального остовного дерева — можно взглянуть под другим углом, что позволит получить еще лучший аппроксимирующий алгоритм для решения задачи коммивояжера. Вспомним, что *эйлеровым циклом* в графе $G$ называется маршрут, который проходит по каждому ребру этого графа только один раз[1]. Узнать, содержит ли связный граф эйлеров цикл, можно, проверив простую характеристику: степень каждой вершины должна быть четной. Очевидно, что условие четности степени вершин является обязательным, поскольку мы должны быть в силах выйти из каждой вершины ровно столько же раз, сколько мы в нее вошли. Но это условие также является и достаточным. Более того, в любом связном графе с вершинами четной степени эйлеров цикл можно легко найти за линейное время.

Эвристический алгоритм для решения задачи коммивояжера можно пересмотреть в терминах эйлеровых циклов. Создадим мультиграф $M$, состоящий из двух копий

[1] См. *разд. 18.7.*

каждого ребра минимального остовного дерева графа $G$. Этот граф с $n$ вершинами и $2(n - 1)$ ребрами должен быть эйлеровым, поскольку степень каждой его вершины вдвое больше, чем степень вершин минимального остовного дерева графа $G$. Любой эйлеров цикл графа $M$ будет определять маршрут, обладающий точно такими же свойствами, что и маршрут обхода в глубину, рассмотренный ранее. Поэтому также должен существовать способ создать укороченный маршрут коммивояжера, стоимость которого будет не более чем вдвое превышать стоимость оптимального маршрута.

Это позволяет предполагать, что, возможно, существует даже лучший аппроксимирующий алгоритм для решения задачи коммивояжера, если можно найти лучший способ обеспечить, чтобы все вершины имели четную степень. Вспомним (см. *разд. 8.5.1*), что *паросочетанием* в графе $G = (V, E)$ называется такое подмножество ребер $E' \subset E$, в котором никакие два ребра из $E'$ не опираются на одну и ту же вершину. Таким образом, добавление множества ребер паросочетания в наш граф повышает степень затрагиваемых вершин на единицу, превращая вершины с нечетной степенью в вершины с четной, а вершины с четной — в вершины с нечетной (рис. 12.4, *б*).

Итак, начнем с идентификации в минимальном остовном дереве графа $G$ вершин с нечетной степенью, которые являются препятствием для нахождения в этом дереве эйлерова цикла. Любой граф должен содержать четное количество вершин с нечетной степенью. Но, добавив между этими вершинами с нечетной степенью множество ребер паросочетания, мы сделаем наш граф эйлеровым. Идеальное паросочетание с минимальной стоимостью (означающее, что каждая вершина должна присутствовать ровно в одном паросочетаемом ребре) поддается эффективному вычислению, как показано в *разд. 18.6*.

Эвристический алгоритм Кристофидеса создает мультиграф $M$, состоящий из минимального остовного дерева графа $G$ плюс множество паросочетаемых ребер минимального веса между вершинами с нечетной степенью этого дерева. Таким образом, мультиграф $M$ является эйлеровым графом и содержит эйлеров цикл, который можно сократить, чтобы создать маршрут коммивояжера весом не более чем $M$.

Обратите внимание на то, что стоимость паросочетания только вершин с нечетной степенью должна быть нижней границей стоимости паросочетания полного графа $G$, — в предположении, что удовлетворяется аксиома треугольника.

На рис. ЦВ-12.5 показано, что чередующиеся ребра любого маршрута коммивояжера должны определять паросочетания (выделены красным и синим цветом), поскольку каждая вершина входит в это множество ребер только один раз.

Стоимость этих красных (или синих) ребер должна быть как минимум такой, что и у паросочетания с минимальным весом графа $G$, а вес (ребер цвета с меньшим весом) — составлять не более половины веса маршрута коммивояжера. То есть стоимость ребер паросочетания, которые мы добавили к мультиграфу $M$, должна быть не более чем половина стоимости оптимального маршрута коммивояжера.

Наконец, общий вес мультиграфа $M$ должен быть максимум в $(1 + (1/2)) = (3/2)$ раза большим, чем вес оптимального маршрута коммивояжера, а это означает, что эвристический алгоритм Кристофидеса создает маршрут весом, не более чем в 3/2 раза превы-

шающим вес оптимального маршрута. Как и в случае с эвристическим алгоритмом для вычисления минимального остовного дерева, потеря веса в результате укорачивания маршрута означает, что полученный маршрут может быть даже лучше, чем гарантированный. Но он никогда не будет хуже.

# 12.4. Когда среднее достаточно хорошее

В вымышленном городке Lake Wobegon все дети ростом выше среднего[2]. Для некоторых задач оптимизации все решения (или их большинство) выглядят близкими к наилучшим возможным. Это обстоятельство позволяет получить очень простые аппроксимирующие алгоритмы с доказуемой гарантией результата, который часто можно улучшить методами эвристического поиска, которые рассматриваются в *разд. 12.6*.

## 12.4.1. Задача максимальной *k*-SAT

В задаче 3-SAT (см. *разд. 11.4.1*) для заданного множества трехэлементных логических дизъюнкций типа:

$$v_3, \text{ или } \overline{v}_{17}, \text{ или } v_{24}$$

требуется вычислить, как присвоить каждой переменной $v_i$ такое значение *истина* или *ложь*, чтобы все дизъюнкции имели значение *истина*.

В более общей задаче *максимальной* 3-SAT требуется вычислить такой набор булевых значений, чтобы максимизировать количество дизъюнкций со значением *истина*. В базовой задаче 3-SAT требуется выяснить, возможно ли присвоить удовлетворяющие значения истинности 100% дизъюнкций, поэтому задача максимальной 3-SAT должна быть сложной. Но сейчас у нас задача оптимизации, поэтому можно подумать о разработке аппроксимирующих алгоритмов ее решения.

Что будет, если каждой переменной $v_i$ присваивать значение по броску монеты и, таким образом, создать полностью произвольный набор значений истинности? Какая доля дизъюнкций будет по нашим ожиданиям удовлетворена таким способом? Возьмем, например, дизъюнкцию, приведенную в начале этого раздела. Она будет удовлетворена во всех случаях, за исключением следующего набора значений истинности: $v_3$ = *ложь*, $\overline{v}_{17}$ = *истина* и $v_{24}$ = *ложь*. Вероятность получить удовлетворяющий набор значений истинности для такой дизъюнкции составляет $1-(1/2)^3 = 7/8$. Таким образом, ожидается, что любой произвольный набор значений истинности будет удовлетворительным для 7/8, или 87,5%, дизъюнкций.

Этот результат выглядит достаточно хорошим для неосмысленного подхода к решению NP-полной задачи. Для экземпляра максимальной задачи *k*-SAT с $m$ входными дизъюнкциями ожидается удовлетворение $m(1-(1/2)^k)$ из этих дизъюнкций с любым произвольным набором значений истинности. С точки зрения аппроксимации чем длиннее дизъюнкции, тем легче получить результат, близкий к оптимальному.

---

[2] См. **https://en.wikipedia.org/wiki/Lake_Wobegon#Standard_monologue_items**.

### 12.4.2. Максимальный бесконтурный подграф

Бесконтурные ориентированные графы легче поддаются обработке, чем общие ориентированные графы. Иногда будет полезным упростить тот или иной граф, удалив набор ребер или вершин, достаточный для разрыва контуров (циклов). Такие задачи — о *разрывающем множестве* дуг (feedback set) — обсуждаются в *разд. 19.11*.

Здесь же мы рассмотрим интересную задачу из этого класса, в которой при разрыве всех ориентированных контуров нужно оставить как можно больше ребер. Постановка задачи выглядит следующим образом:

**Задача**. Найти максимальный бесконтурный ориентированный подграф.

**Вход.** Ориентированный граф $G = (V, E)$.

**Выход.** Самое большое подмножество $E' \subseteq E$, для которого $G' = (V, E')$ не содержит контуров.

Существует очень простой алгоритм, который гарантирует решение с количеством ребер равным, по крайней мере, половине оптимального. Я рекомендую вам попробовать разработать такой алгоритм самостоятельно, прежде чем вы прочитаете его описание.

Итак, создаем *любую* перестановку вершин и рассматриваем ее как упорядочивание слева направо — аналогично топологической сортировке. Теперь некоторые ребра будут направлены слева направо, в то время как оставшиеся ребра — справа налево.

Размер любого из этих подмножеств ребер должен быть, по крайней мере, таким же, как другого. Это означает, что оно содержит хотя бы половину ребер. Кроме того, каждое из этих двух подмножеств ребер должно быть бесконтурным, поскольку только бесконтурные ориентированные графы можно подвергать топологической сортировке, — контур (цикл) нельзя создать, постоянно двигаясь в одном направлении. Таким образом, подмножество ребер большего размера должно быть бесконтурным и содержать, по крайней мере, половину ребер оптимального решения.

Этот аппроксимирующий алгоритм чрезвычайно прост. Но обратите внимание, что на практике применение эвристики может улучшить его производительность, при этом сохраняя гарантию качества результата. Возможно, имеет смысл попробовать несколько произвольных перестановок и выбрать ту, которая дает наилучший результат. Кроме того, можно попытаться обменивать местами пары вершин в перестановках и оставлять те варианты обмена, которые добавляют большее количество ребер в подмножество большего размера.

## 12.5. Задача о покрытии множества

Предыдущие разделы могут создать ложное представление, что с помощью аппроксимирующих алгоритмов возможно получить решение любой задачи с константным коэффициентом отличия от оптимального решения. Однако решение некоторых задач из каталога, приведенного в *части II* книги, — например, задачи поиска максимальной клики, нельзя аппроксимировать с *любым* наперед заданным коэффициентом.

*Задача о покрытии множества* занимает промежуточную позицию между этими двумя крайностями, поскольку имеет алгоритм, выдающий решение с коэффициентом от-

личия от оптимального, равным $\Theta(\lg n)$. Задача о покрытии множества представляет собой общий случай задачи о вершинном покрытии. Формальная постановка задачи (рассматриваемой в *разд. 21.1*) выглядит следующим образом:

**Задача**. Найти покрытие множества.

**Вход.** Семейство подмножеств $S = \{S_1, ..., S_m\}$ универсального множества $U = \{1, ..., n\}$.

**Выход.** Подмножество наименьшей мощности $T$ семейства $S$, чье объединение равно универсальному множеству, — т. е. $\bigcup_{i=1}^{|T|} T_i = U$.

Вполне естественно применить для решения этой задачи эвристический алгоритм «жадного» типа. В частности, постоянно выбирать подмножество, которое покрывает наибольшее подсемейство непокрытых на текущем этапе элементов, пока не получим полное покрытие. Псевдокод соответствующего алгоритма показан в листинге 12.3.

**Листинг 12.3. Аппроксимирующий алгоритм поиска покрытия множества**

```
SetCover(S)
  While (U ≠ 0) do:
    Определяем подмножество Si, имеющее наибольшее пересечение
    с множеством U
    Выбираем Si для покрытия множества
    U = U − Si
```

Одним из побочных эффектов этого процесса выбора является тот факт, что по мере исполнения алгоритма количество покрытых на каждом шаге элементов образует невозрастающую последовательность. Почему? Потому что в противном случае «жадный» алгоритм раньше выбрал бы более мощное подмножество, если бы оно действительно существовало.

Таким образом, этот эвристический алгоритм можно рассматривать как уменьшение количества непокрытых элементов от $n$ до нуля, причем невозрастающими порциями. Пример трассировки исполнения такого алгоритма показан на рис. 12.6.

| Промежуточный этап | 6 | 5 | | 4 | | | | | 3 | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Непокрытые элементы | 64 | 51 | 40 | 30 | 25 | 22 | 19 | 16 | 13 | 10 | 7 | 4 | 2 | 1 |
| Размер выбранного подмножества | 13 | 11 | 10 | 5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 1 | 1 |

**Рис. 12.6.** Работа «жадного» алгоритма на экземпляре задачи о покрытии множества. Ширина $w$ определяется пятью подмножествами на промежуточном этапе 4, когда количество не входящих в покрытие элементов уменьшится наполовину с не менее чем $2^5-1$ до не более чем $2^4$

Важное контрольное событие этой трассировки происходит, как только количество оставшихся непокрытых элементов уменьшается в $2^n$ раз. Очевидно, что таких событий может быть самое большее $\lceil \lg n \rceil$.

Пусть $w_i$ обозначает количество подмножеств, выбранных нашим эвристическим алгоритмом для покрытия элементов между контрольными событиями, соответствующими

уменьшению в $2^{i+1} - 1$ раз и $2^i$ раз. Определим максимальную ширину $w_i$ как $w$, где $0 \le i \le \lg n$. В примере трассировки, прведенном на рис. 12.6, максимальная ширина представлена пятью подмножествами, необходимыми для перехода от $2^5 - 1$ к $2^4$.

Так как существует самое большее $\lg n$ таких контрольных событий, то выдаваемое «жадным» эвристическим алгоритмом решение должно содержать самое большее $w \lg n$ подмножеств. Но я утверждаю, что оптимальное решение должно содержать, *по крайней мере*, $w$ подмножеств, так что выдаваемое эвристическим алгоритмом решение хуже оптимального не больше чем в $\lg n$ раз.

Почему? Рассмотрим среднее количество новых элементов, покрываемых при проходе между контрольными событиями, соответствующими уменьшению в $2^{i+1} - 1$ раз и $2^i$ раз. Для этих $2^i$ элементов требуется $w_i$ подмножеств, поэтому мощность среднего покрытия $\mu_i = 2^i/w_i$. Более того, последнее (наименьшее) из этих подмножеств покрывает самое большее $\mu_i$ подмножеств. Таким образом, множество $S$ не содержит подмножества, которое может покрыть больше чем $\mu_i$ из оставшихся $2^i$ элементов. Поэтому, чтобы получить решение, нам нужно, по крайней мере, $2^i/\mu_i = w_i$ подмножеств.

Несколько неожиданным является то обстоятельство, что на самом деле существуют экземпляры покрытия множества, для которых этот эвристический алгоритм находит решения величиной в $\Omega(\lg n)$ раз больше оптимального, как, например, показано на рис. 12.2. Этот логарифмический множитель является свойством задачи и эвристического алгоритма для ее решения, а не следствием плохо проведенного анализа.

#### Подведение итогов

Аппроксимирующие алгоритмы гарантируют решения, которые всегда близки к оптимальному. Они могут открыть практический подход к решению NP-полных задач.

## 12.6. Эвристические методы поиска

Отслеживание с возвратом дало нам способ найти лучшее из всех возможных решений, оцененное по заданной целевой функции.

Метод перебора с возвратом позволяет нам найти *наилучшее* из возможных рещений, оцененное по заданной целевой функции. Но любой алгоритм, исследующий все возможные конфигурации, обречен быть невероятно дорогим на входных экземплярах большого размера. Эвристические методы предоставляют альтернативный подход к оптимизации трудных комбинаторных задач.

В этом разделе мы рассмотрим такие эвристические методы поиска. Основное внимание здесь уделено методу имитации отжига, который я считаю наиболее надежным для практического применения. Эвристические поисковые алгоритмы поначалу кажутся каким-то шаманством, но если их подвергнуть тщательному исследованию, то выяснится, что многое в их работе вполне поддается логическому объяснению.

Мы рассмотрим три разных эвристических метода поиска: *произвольную выборку* (random sampling), *градиентный спуск* (gradient descent) и *имитацию отжига* (simulated annealing). Для сравнения этих трех эвристических методов мы испытаем их на задаче коммивояжера. Все три метода имеют два общих компонента:

- *представление кандидатов решения.*

  Это полное, но при этом краткое описание возможных решений задачи, используемое в точности так же, как и при поиске перебором с возвратом. Для задачи коммивояжера пространство решений содержит $(n-1)!$ элементов, т. е. все возможные циклические перестановки вершин. Для представления каждого элемента пространства решений нам требуется соответствующая структура данных. Для задачи коммивояжера решения-кандидаты можно представлять с помощью массива $S$, содержащего $n-1$ вершин, где ячейка $S_i$ определяет $(i+1)$-ю вершину маршрута, начинающегося в вершине $v_1$;

- *функция стоимости.*

  Поисковые алгоритмы должны иметь функцию *стоимости* или *оценки*, чтобы определять качество каждого возможного решения. Наш эвристический алгоритм поиска определяет элемент с наилучшей оценкой: наибольшим или наименьшим значением — в зависимости от природы задачи. В случае задачи коммивояжера функция стоимости для оценки кандидатов решения $S$ лишь суммирует вес всех ребер $(S_i, S_{i+1})$, где как $S_0$, так и $S_n$ обозначают $v_1$.

## 12.6.1. Произвольная выборка

Самым простым подходом к организации поиска в пространстве решений является произвольная выборка, называемая также *методом Монте-Карло*. В этом случае последовательно создаются и оцениваются произвольные решения, и процесс прекращается, как только будет найдено достаточно хорошее решение или когда нам надоест ждать его получения (что более вероятно). В качестве конечного решения выбирается самое лучшее решение изо всех исследованных в процессе выборки.

Для действительно произвольной выборки элементы необходимо выбирать из пространства решений *равномерно произвольным образом*. Это означает, что вероятность выбора любого элемента из пространства решений в качестве следующего кандидата должна быть одинаковой. Реализация такой выборки может быть сложнее, чем может показаться на первый взгляд. Алгоритмы для генерирования случайных перестановок, подмножеств, разбиений и графов рассматриваются в *разд. 17.4–17.7*. А в листинге 12.4 предоставлена процедура произвольного выбора решений.

**Листинг 12.4. Процедура произвольного выбора решений**

```
void random_sampling(tsp_instance *t, int nsamples, tsp_solution *s) {
    tsp_solution s_now;  /* Текущее решение задачи коммивояжера */
    double best_cost;    /* Самая лучшая стоимость в настоящий момент */
    double cost_now;     /* Текущая стоимость */
    int i;               /* Счетчик */

    initialize_solution(t->n, &s_now);
    best_cost = solution_cost(&s_now, t);
    copy_solution(&s_now, s);

    for (i = 1; i <= nsamples; i++) {
        random_solution(&s_now);
```

```
        cost_now = solution_cost(&s_now, t);
        if (cost_now < best_cost) {
            best_cost = cost_now;
            copy_solution(&s_now, s);
        }

        solution_count_update(&s_now, t);
    }
}
```

В каких случаях можно успешно использовать произвольную выборку?

- *Пространство решений содержит большую долю приемлемых решений.*

  Найти травинку в стоге сена намного легче, чем иголку. Когда существует много хороших решений, то произвольная выборка должна быстро привести к одному из них.

  Одним из примеров успешного применения метода произвольной выборки является поиск простых чисел. Построение больших случайных простых чисел для ключей — важный аспект криптографических систем, таких как, например, RSA-кодирование. Простым будет приблизительно каждое число $\ln n$, поэтому, чтобы найти простое число длиной в несколько сотен цифр, требуется выполнить умеренное количество произвольных выборок.

- *Пространство решений не является однородным.*

  Произвольную выборку нужно применять в тех случаях, когда отсутствуют какие бы то ни было признаки приближения к решению. Допустим, что вам надо выбрать среди своих друзей того, номер полиса социального страхования которого заканчивается на 00. Для решения этой задачи не существует другого метода, кроме как спросить произвольно выбранного товарища, каков номер его страховки.

  Возвратимся опять к задаче поиска больших простых чисел. Эти числа разбросаны среди других целых чисел без какой-либо системы. Использование произвольной выборки для их поиска будет не хуже любого другого метода.

Но подходит ли метод произвольной выборки для решения задачи коммивояжера? Нет. Самым лучшим решением задачи коммивояжера, которое я нашел, проверив 100 миллионов произвольных перестановок экземпляра задачи коммивояжера со 150 узлами, было 43 251 единица. Это более чем в восемь раз превышало длину оптимального маршрута! Пространство решений этой задачи почти полностью состоит из посредственных и плохих решений, поэтому качество решений с ростом количества выборок, поделенного на время поиска, растет очень медленно. Чтобы дать представление о разбросе решений между выборками, на рис. 12.7 показаны колебания результатов произвольно выбираемых решений (как правило, низкого качества).

Подобно задаче коммивояжера, большинство встречающихся задач имеют сравнительно небольшое количество хороших решений и высокую степень однородности пространства решений. Для эффективного решения таких задач — типа поиска иголки в стогу сена — требуются более мощные эвристические алгоритмы.

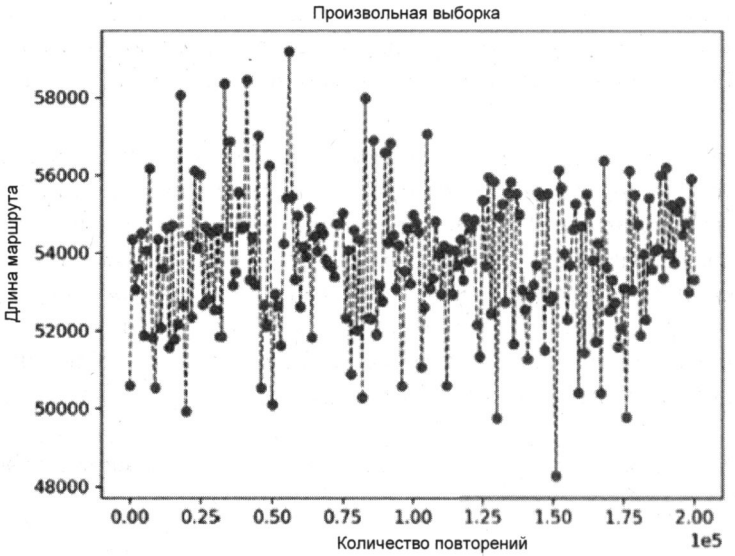



**Рис. 12.7.** Соотношения между временем поиска решений и их качеством при решении задачи коммивояжера методом произвольной выборки. Процесс нахождения каждого нового наилучшего (здесь — наименьшего) решения идет очень медленно

## ОСТАНОВКА ДЛЯ РАЗМЫШЛЕНИЙ: Выбор пары

**ЗАДАЧА.** Требуется выбрать две произвольные вершины графа и поменять их местами. Предложите эффективный алгоритм для генерирования произвольных элементов с равномерным распределением из $\binom{n}{2}$ неупорядоченных пар множества $\{1, ..., n\}$.

**РЕШЕНИЕ.** Рассмотрим следующую процедуру для генерирования случайных неупорядоченных пар чисел:

```
i = random_int(1, n-1);
j = random_int(i+1, n);
```

Ясно, что эта процедура на самом деле генерирует неупорядоченные пары, т. к. $i < j$. Кроме этого, если допустить, что функция `random_int` генерирует целые числа в диапазоне своих двух аргументов однородно, также ясно, что можно сгенерировать все $\binom{n}{2}$ неупорядоченные пары.

Будет ли распределение этих пар однородным? Нет. Какова вероятность генерирования пары (1, 2)? Вероятность получения 1 равна $1/(n-1)$, а вероятность получения 2 также равна $1/(n-1)$, что дает нам вероятность получения пары $p(1, 2) = 1/(n-1)^2$. Но какова вероятность получения пары чисел $(n-1, n)$? Опять же, вероятность получения первого числа равна $1/(n-1)$, но для второго кандидата пары существует только один возможный вариант! Эта пара будет выпадать в $n-1$ раз чаще, чем (1, 2).

Проблема заключается в том, что количество пар, содержащих первое большое число, меньше, чем пар, содержащих первое малое число. Проблему можно было бы решить,

вычислив, сколько именно неупорядоченных пар начинаются с числа $i$ (ровно $n - i$), и соответствующим образом скорректировав вероятность. Тогда второе число пары можно было бы выбирать произвольным образом в диапазоне от $i + 1$ до $n$ с равномерным распределением.

Но вместо того, чтобы заниматься вычислениями, лучше воспользоваться тем обстоятельством, что генерирование $n^2$ *упорядоченных* пар — это весьма простая задача. Произвольно выбираем два целых числа независимо друг от друга. Игнорируя упорядоченность, превращая упорядоченную пару в неупорядоченную $(x, y)$, где $x < y$, мы получаем вероятность генерирования каждой пары разных цифр, равную $2/n^2$. В случае генерирования пары $(x, x)$ она отбрасывается и генерируется новая пара. Алгоритм для генерирования равномерно распределенных произвольных пар целых чисел с ожидаемым постоянным временем исполнения приводится в листинге 12.5.

**Листинг 12.5. Генерирование равномерно распределенных произвольных пар целых чисел**

```
do {
   i = random_int(1,n);
   j = random_int(1,n);
   if (i > j) swap(&i,&j);
} while (i==j);
```

■

## 12.6.2. Локальный поиск

Допустим, нам нужно найти эксперта по алгоритмам для решения определенной задачи. Мы *можем* позвонить по случайному телефонному номеру и спросить ответившего, не является ли он таким специалистом. В случае отрицательного ответа мы вешаем трубку и повторяем процесс, пока не найдем нужного нам профессионала. После многократных звонков мы, в конце концов, найдем нужного человека, но было бы намного эффективнее спросить первого ответившего, нет ли среди его знакомых того, кто, возможно, является экспертом по алгоритмам, и следующий звонок сделать уже *ему*.

Эта стратегия сканирования *прилегающего пространства* вокруг каждого элемента пространства решений называется *локальным поиском*. Каждый такой кандидат решения $x$ можно рассматривать как вершину с исходящим ребром $(x, y)$, направленным к каждому другому кандидату на решение $y$, являющемуся соседом $x$. Поиск выполняется из вершины $x$ по направлению к наиболее перспективному кандидату вблизи этой вершины.

Ясное дело, что мы *не хотим* создавать граф этого района для пространства решений большого размера. Представьте себе задачу коммивояжера, которая в таком графе будет иметь $(n - 1)!$ вершин. Так что мы ищем решение посредством эвристического алгоритма, поскольку не надеемся решить задачу в течение приемлемого времени исчерпывающим перебором всех вариантов.

На самом деле, мы хотим получить механизм перехода к близлежащему решению, слегка модифицировав текущее. Типичные механизмы перехода включают обмен мес-

тами произвольной пары элементов или изменение (вставку или удаление) одного элемента решения.

Подходящим механизмом перехода для задачи коммивояжера стал бы обмен местами произвольной пары вершин $S_i$ и $S_j$ текущего маршрута, как показано на рис. 12.8. Такой обмен вершин изменяет до восьми ребер маршрута, удаляя четыре ребра, смежные с $S_i$ и $S_j$, и добавляя другие. Эффект этих инкрементальных изменений на качество решения можно вычислить также инкрементным образом, вследствие чего время исполнения функции оценки стоимости будет пропорционально размеру изменений (обычно постоянному), что станет большим улучшением по сравнению с линейным размером решения. Еще лучшим вариантом было бы заменить два *ребра* маршрута двумя другими, поскольку это может облегчить нахождение шагов, улучшающих стоимость маршрута.

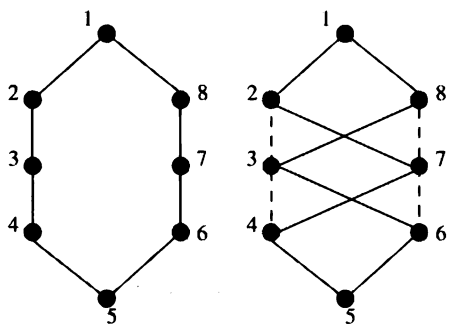



**Рис. 12.8.** Улучшение маршрута коммивояжера за счет обмена местами вершин 3 и 7 с заменой четырех старых ребер маршрута четырьмя новыми

Эвристический алгоритм локального поиска начинает работу с произвольного элемента пространства решений и сканирует смежную с ним область, пытаясь найти подходящего кандидата на выполнение перехода. При благоприятной замене вершин стоимость четырех вставленных вершин — согласно вычислениям, выполняемым функцией перехода (`transition`), — меньше стоимости удаленных. А для задачи другого типа, называемой *восхождением по выпуклой поверхности* (hill-climbing), мы пытаемся найти самую высшую (или самую низшую) точку, начав поиск в произвольной точке и выбирая любой промежуточный маршрут, ведущий в требуемом направлении. Эта процедура выбора промежуточного отрезка маршрута повторяется до тех пор, пока не дойдет до точки, в которой все исходящие направления не удовлетворяют нашим требованиям (листинг 12.6). Теперь вы «Царь горы» (или — для задачи минимизации — «Король канавы»).

**Листинг 12.6. Процедура восхождения по выпуклой поверхности**

```
void hill_climbing(tsp_instance *t, tsp_solution *s) {
    double cost;          /* Самая лучшая стоимость в настоящий момент */
    double delta;         /* Стоимость обмена */
    int i, j;             /* Счетчики */
    bool stuck;           /* Это решение лучшее? */
```

```
    initialize_solution(t->n, s);
    random_solution(s);
    cost = solution_cost(s, t);

    do {
        stuck = true;
        for (i = 1; i < t->n; i++) {
            for (j = i + 1; j <= t->n; j++) {
                delta = transition(s, t, i, j);
                if (delta < 0) {
                    stuck = false;
                    cost = cost + delta;
                } else {
                    transition(s, t, j, i);
                }
                solution_count_update(s, t);
            }
        }
    } while (stuck);
}
```

Но, к сожалению, скорее всего, вы не «Царь горы». И вот почему. Допустим, что вы, проснувшись утром в отеле на горнолыжном курорте, решили начать свое восхождение на гору. Ближайшей высшей для вас точкой окажется верхний этаж отеля, а потом его крыша. И дальше вам идти некуда. Поскольку, чтобы взобраться на вершину горы, вам нужно сначала выйти на улицу, для чего придется спуститься на первый этаж отеля. Но это нарушает требование, что каждый шаг должен повышать ваше местонахождение. Алгоритм восхождения по выпуклой поверхности и подобные эвристические алгоритмы, такие как «жадный» поиск или локальный поиск, позволяют быстро получить оптимальные локальные результаты, но часто не справляются с поиском наилучшего глобального решения.

В каких случаях локальный поиск приводит к успеху?

- *Пространство решений является высокооднородным.*

  Метод восхождения по выпуклой поверхности лучше всего работает с *выпуклым* пространством решений — т. е. с пространством, содержащим ровно одно возвышение. Таким образом, независимо от местонахождения в пространстве решений точки начала поиска, у нас всегда имеется направление, в котором нужно продолжать поиск до тех пор, пока мы не дойдем до глобального максимума.

  Этим свойством обладают многие естественные задачи. В частности, можно считать, что двоичный поиск начинается в середине пространства решений. В этой точке существует только одно из двух возможных направлений, в котором нужно идти, чтобы приблизиться к целевому элементу. Симплексный алгоритм для линейного программирования (см. *разд. 16.6*) представляет собой не что иное, как восхождение по выпуклой поверхности правильного пространства решений, но тем не менее гарантирует получение оптимального решения для любой задачи линейного программирования.

- *Стоимость оценки изменения намного ниже стоимости глобальной оценки.*

  Стоимость оценки произвольного решения из $n$ вершин для задачи коммивояжера равна $\Theta(n)$, поскольку нам нужно суммировать стоимость каждого ребра в циклической перестановке, описывающей маршрут. Но когда эта стоимость известна, то стоимость маршрута после обмена местами той или иной пары вершин можно определить за постоянное время.

  Если у нас имеется очень большое значение $n$ и ограниченное время для поиска, то лучше потратить это время на выполнение нескольких инкрементальных оценок, чем на сколько-то произвольных выборок, даже если мы ищем иголку в стоге сена.

Основным недостатком локального поиска является ситуация, при которой как только мы нашли локальный оптимум, не остается больше никаких вариантов для поиска глобального решения. Конечно же, если у нас имеется время, мы можем начать новый поиск с другой произвольно выбранной точки, но в поисковом пространстве, содержащем много небольших возвышенностей, маловероятно найти оптимальное решеие.

Насколько хорошим является метод локального поиска для решения задачи коммивояжера? Намного лучшим, чем произвольная выборка, при сходных временных затратах. Длина наилучшего маршрута, полученного локальным поиском на нашем сложном экземпляре задачи коммивояжера из 150 узлов, получилась равной 15 715 единиц, что почти в три раза лучше, чем результат поиска методом произвольной выборки[3].

На рис. 12.9 показаны результаты применения метода локального поиска — многократные переходы от случайных маршрутов к удовлетворительным решениям примерно одинакового качества.

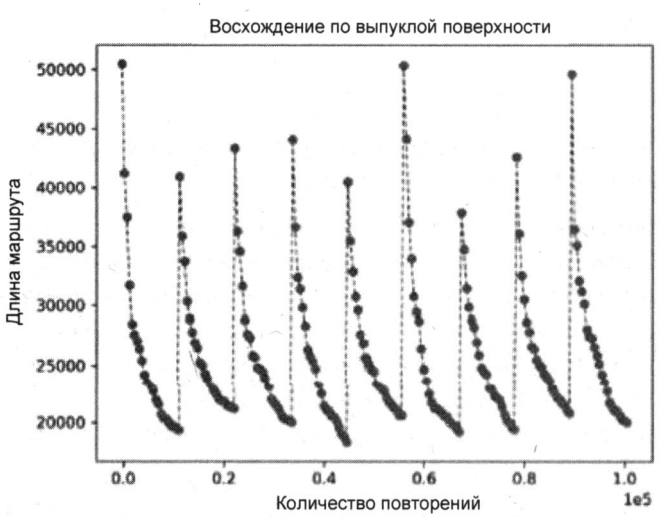



**Рис. 12.9.** Отношение времени поиска к качеству решений задачи коммивояжера для метода восхождения по выпуклой поверхности

[3] Но это все равно больше чем в два раза превышает стоимость оптимального решения, равного 6828 единиц, поэтому решение, полученное посредством аппроксимации минимального остовного дерева (см. *разд. 12.3*), было бы наилучшим.

Хотя этот результат намного лучше, чем полученный методом произвольных выборок, действительно хорошим его назвать трудно. К примеру, были бы вы довольны, если бы вам пришлось платить вдвое больше налогов, чем вы в действительности должны? Вывод: нам нужны более мощные методы для получения решения, близкого к оптимальному.

## 12.6.3. Имитация отжига

*Имитация отжига* (simulated annealing) представляет собой эвристическую процедуру поиска, допускающую случайные переходы, что ведет к более дорогостоящим (и соответственно худшим) решениям. Это выглядит как шаг назад, но позволяет удержать поиск от зацикливания на оптимальном локальном решении. Если вернуться к нашей аналогии с покорением горных вершин, начиная с номера в горнолыжном отеле, можно сказать, что теперь нам разрешается сойти вниз по лестнице или выпрыгнуть в окно, а затем продолжить восхождение в правильном направлении.

Идея имитации отжига основана на аналогии с физическим процессом остывания расплавленных материалов и последующим их переходом в твердое состояние. В теории термодинамики энергетическое состояние системы определяется энергетическим состоянием каждой составляющей его частицы. Но частица переходит из одного энергетического состояния в другое произвольным образом, при этом переходы между состояниями обусловливаются температурой системы.

В частности, вероятность обратного перехода $P(e_i, e_j, T)$ из низкоэнергетического (высокого качества) состояния $e_i$ в высокоэнергетическое (низкого качества) состояние $e_j$ при температуре $T$ определяется формулой

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)},$$

где $k_B$ — положительная постоянная Больцмана, используемая для настройки требуемой частоты ходов в обратном направлении.

Каков смысл этой формулы? Переход из низкоэнергетического состояния в высокоэнергетическое состояние: $e_i - e_j < 0$ — подразумевает отрицательный показатель степени. Обратите внимание, что $0 \leq e^{-x} = 1/e^x \leq 1$ для любого положительного значения $x$. В результате мы имеем вероятность, которая уменьшается с увеличением $|e_i - e_j|$. Таким образом, существует ненулевая вероятность перехода в высокоэнергетическое (низкого качества) состояние, причем переходы малой амплитуды вероятнее, чем значительной. Кроме того, чем выше температура $T$, тем выше вероятность таких энергетических переходов.

Какое отношение все это имеет к комбинаторной оптимизации? При охлаждении физическая система стремится достичь состояния с минимальной энергией. Минимизация полной энергии представляет собой задачу комбинаторной оптимизации для любого набора дискретных частиц. Посредством произвольных переходов, генерируемых согласно заданному распределению вероятностей, мы можем эмулировать физические процессы, чтобы решать произвольные задачи комбинаторной оптимизации. Псевдокод алгоритма для такой эмуляции приводится в листинге 12.7.

**Листинг 12.7. Алгоритм имитации отжига**

```
Simulated-Annealing()
    Создаем первоначальное решение s
    Инициализируем температуру T
    repeat
        for i = 1 to iteration-length do
            Произвольно выбираем s_i соседом s
            if (C(s) ≥ C(s_i)) then s = s_i
            else if (e^((C(s)-C(s_i))/(k_B·T)) > random[0,1]) then s = s_i
        Понижаем температуру T
    until (no change in C(s))
    Return s
```

### Подведение итогов

**Имитация отжига является эффективным средством, потому что она больше времени уделяет «хорошим» элементам пространства решений, чем «плохим», а также потому, что она не зацикливается на одном локальном оптимальном решении.**

Так же как и в случае с локальным поиском, представление задачи состоит из представления пространства решений и задания несложной функции стоимости $C(s)$ для определения качества конкретного решения. Новым компонентом является *график охлаждения* (cooling schedule), параметры которого управляют вероятностью приемлемости «плохого» перехода в зависимости от времени.

В начале поиска мы стремимся использовать фактор случайности, чтобы исследовать пространство поиска, поэтому вероятность приемлемости «плохого» перехода должна быть высокой. В процессе поиска мы стремимся ограничиться переходами к локальным улучшениям и оптимизациям. Упомянутый график охлаждения можно регулировать с помощью следующих параметров:

- *первоначальная температура системы* — обычно $T_i = 1$;
- *функция понижения температуры* — обычно $T_k = \alpha \cdot T_{k-1}$, где $0{,}8 \leq \alpha \leq 0{,}99$. Это означает экспоненциальное понижение температуры, а не линейное;
- *количество итераций перед понижением температуры* — как правило, разрешается провести около 1000 итераций. Кроме этого, обычно весьма выгодно удерживать заданную температуру в течение многократных итераций, пока здесь наблюдается прогресс;
- *критерии приемлемости* — в типичном случае приемлем любой «хороший» переход, а также «плохой» переход, если

$$e^{\frac{C(s_{i-1})-C(s_i)}{k_B T}} > r,$$

где $r$ — случайное число в диапазоне $0 \leq r < 1$. Константа Больцмана $k_B$ масштабирует эту функцию стоимости, чтобы при начальной температуре принимались почти все переходы;

*критерии остановки* — если за последнюю итерацию (или несколько последних итераций) значение текущего решения не изменилось или не улучшилось, то поиск прекращается и выводится текущее решение.

Создание правильного графика охлаждения фактически выполняется методом проб и ошибок в процессе подстановки разных значений констант и наблюдения за результатами. Я рекомендую начинать работу с методом имитации отжига, ознакомившись с уже существующими его реализациями. Для этого поэкспериментируйте с моей полной реализацией этого метода, которую можно найти на сайте **http://www.algorist.com**.

Сравните соотношения между временем поиска решений и их качеством для всех трех рассмотренных эвристических алгоритмов. Наилучшую эффективность демонстрирует алгоритм имитации отжига. На рис. 12.10 показаны три прогона для трех разных произвольных инициализаций, напоминающих кардиограмму останавливающегося сердца, когда они сходятся к минимальному значению. Поскольку они не застревают на локальном оптимальном решении, все три прогона дают намного лучшие решения, чем самое лучшее решение методом восхождения по выпуклой поверхности. Более того, чтобы получить большее улучшение результата, требуется сравнительно небольшое количество итераций, как можно видеть по трем резким падениям диапазона решений к оптимальному значению.

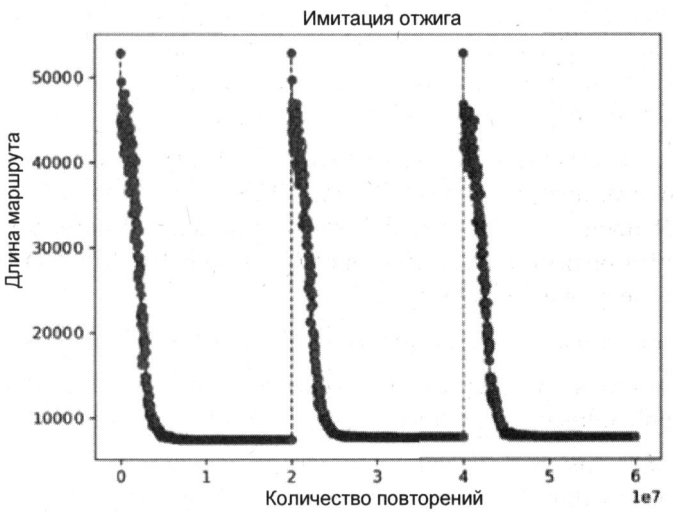



**Рис. 12.10.** Соотношения между временем поиска решений и их качеством для решения задачи коммивояжера методом имитации отжига

После десяти миллионов итераций метод имитации отжига выдал решение стоимостью 7212 единицы, что всего лишь на 10,4% превышает оптимальное. Если вы готовы немного подождать, то можно получить еще более качественное решение. В частности, выполнение одного миллиарда итераций (что на моем ноутбуке заняло только 5 минут и 21 секунду) понизило стоимость решения до 6850 единиц, что всего лишь на 4,9% больше оптимальной.

При умелом обращении самые лучшие эвристические алгоритмы, настроенные под решение задачи коммивояжера, могут выдать чуть более удачное решение, чем метод имитации отжига. Однако в нашем случае метод имитации отжига работает превосходно, не требуя специальной настройки. Я предпочитаю для решения задач оптимизации пользоваться именно им.

## Реализация

Реализация эвристического алгоритма имитации отжига приведена в листинге 12.8.

**Листинг 12.8. Реализация метода имитации отжига**

```
void anneal(tsp_instance *t, tsp_solution *s) {
    int x, y;                       /* Пара элементов для обмена местами */
    int i, j;                       /* Счетчики */
    bool accept_win, accept_loss; /* Условия принятия перехода */
    double temperature;             /* Текущая температура системы */
    double current_value;           /* Значение текущего состояния */
    double start_value;             /* Значение в начале цикла */
    double delta;                   /* Значение после обмена */
    double exponent;                /* Показатель степени для функции
                                                энерг. состояния */

    temperature = INITIAL_TEMPERATURE;

    initialize_solution(t->n, s);
    current_value = solution_cost(s, t);

    for (i = 1; i <= COOLING_STEPS; i++) {
        temperature *= COOLING_FRACTION;

        start_value = current_value;

        for (j = 1; j <= STEPS_PER_TEMP; j++) {
              /* Выбираем индексы элементов для обмена местами */
            x = random_int(1, t->n);
            y = random_int(1, t->n);

            delta = transition(s, t, x, y);
            accept_win = (delta < 0);          /* Обмен понизил
                                                  стоимость? */

            exponent = (-delta / current_value) / (K * temperature);
            accept_loss = (exp(exponent) > random_float(0,1));

            if (accept_win || accept_loss) {
                current_value += delta;
            } else {
                transition(s, t, x, y);            /* Обратный переход */
            }
```

```
            solution_count_update(s, t);
        }

        if (current_value < start_value) {      /* Возвращаемся при этой
                                                   температуре */
            temperature /= COOLING_FRACTION;
        }
    }
}
```

## 12.6.4. Применение метода имитации отжига

Теперь рассмотрим несколько примеров, демонстрирующих, как можно использовать аккуратное моделирование представления состояния и функции стоимости для решения реальных задач комбинаторного поиска.

### Задача максимального разреза

В задаче *максимального разреза* требуется разделить вершины взвешенного графа $G$ на множества $V_1$ и $V_2$ таким образом, чтобы максимизировать в каждом множестве вес (или количество) ребер с одной вершиной. Для графов, представляющих электронные схемы, максимальный разрез графа определяет наибольший поток данных, который может одновременно протекать по схеме. Задача максимального разреза является NP-полной (см. *разд. 19.6*).

Как можно сформулировать задачу максимального разреза для решения методом имитации отжига? Пространство решений состоит из всех $2^{n-1}$ возможных разбиений вершин. Мы получаем двойную экономию по всем подмножествам вершин, зафиксировав вершину $v_1$ на левой стороне разбиения. Подмножество вершин, сопровождающее эту вершину, можно представить с помощью битового вектора. Стоимостью решения является сумма весов разреза в текущей конфигурации. Механизм естественного перехода выбирает произвольным образом одну вершину и перемещает ее в другую часть разбиения, просто изменив на обратное значение соответствующего бита в битовом векторе. Изменение в функции стоимости составит разность между весом старых соседей вершины и весом ее новых соседей. Она вычисляется за время, пропорциональное степени вершины.

На практике именно к такому простому и естественному моделированию следует стремиться при поиске эвристического алгоритма.

### Независимое множество

*Независимым множеством* графа $G$ называется подмножество вершин $S$, не содержащее ребер, у которых обе конечные точки являются членами $S$. Максимальным независимым множеством графа является такое множество вершин, которое порождает пустой (т. е. не содержащий ребер) подграф. Задача поиска больших независимых множеств возникает в задачах рассеивания, связанных с календарным планированием и теорией шифрования (см. *разд. 19.2*).

Естественное пространство состояний для решения задачи методом имитации отжига включает в себя все $2^n$ возможных подмножеств вершин, представленных в виде бито-

вого вектора. Так же как и в случае с максимальным разрезом, простой механизм переходов добавляет или удаляет одну вершину из множества $S$.

Одной из наиболее естественных целевых функций для подмножества $S$ будет функция, возвращающая 0, если порожденный множеством $S$ подграф содержит ребро, и $|S|$ — в случае действительно независимого множества. Такая функция гарантирует, что мы непрерывно приближаемся к получению независимого множества. Но это условие настолько жесткое, что существует большая вероятность остаться в пределах лишь небольшой части общего пространства поиска. Бóльшую степень гибкости и ускорение работы целевой функции можно получить, разрешив непустые графы на ранних этапах охлаждения. Этого можно добиться, используя целевую функцию наподобие $C(S) = |S| - \lambda \cdot m_S/T$, где $\lambda$ является постоянной, $T$ представляет температуру, а $m_S$ — количество ребер в подграфе, порожденном $S$. Для такой цели хорошо подходят большие подмножества с небольшим количеством ребер, а зависимость $C(S)$ от $T$ обеспечивает в конечном итоге вытеснение ребер по мере охлаждения системы.

## Размещение компонентов на печатной плате

Задача разработки печатных плат заключается в размещении на них должным образом компонентов — обычно это интегральные схемы. Заданными критериями компоновки могут быть минимизация площади или отношения длины платы к ее ширине, чтобы она помещалась в отведенное место, и минимизация длины соединяющих дорожек. Компоновка печатных плат представляет собой типичный пример запутанной задачи оптимизации со многими критериями. Для решения таких задач идеально подходит метод имитации отжига.

При формальной постановке задается коллекция прямоугольных модулей $r_1, ..., r_n$ с соответствующими размерами $h_i \times l_i$. Кроме этого, для каждой пары модулей $(r_i, r_j)$ задается количество соединяющих их дорожек $w_{ij}$. Требуется найти такое размещение прямоугольников, которое минимизирует площадь печатной платы и длину соединяющих дорожек, — при условии, что прямоугольники не могут частично или полностью накладываться друг на друга.

Пространство состояний для этой задачи должно описывать расположение на плате каждого прямоугольника. Чтобы сделать задачу дискретной, можно наложить ограничение, при котором прямоугольники могут размещаться только на вершинах решетки целых чисел. Подходящим механизмом переходов может стать перемещение одного прямоугольника в другое место или обмен местами двух прямоугольников. Естественная функция стоимости может выглядеть следующим образом:

$$C(S) = \lambda_{площадь}(S_{высота} \cdot S_{ширина}) + \sum_{i=1}^{n}\sum_{j=1}^{n}(\lambda_{дорожка} w_{ij} \cdot d_{ij} + \lambda_{наложение}(r_i \cap r_j)),$$

где $\lambda_{площадь}$, $\lambda_{дорожка}$ и $\lambda_{наложение}$ являются весами, представляющими влияние этих факторов на функцию стоимости. По всей видимости, значение $\lambda_{наложение}$ должно быть обратно пропорционально температуре, чтобы после предварительного размещения прямоугольников их позиции корректировались во избежание наложения прямоугольников друг на друга.

***Подведение итогов***

Метод имитации отжига является простым, но эффективным методом получения хотя и не оптимальных, но достаточно хороших решений задач комбинаторного поиска.

# 12.7. История из жизни. Только это не радио

— Считайте, что это радио, — мой собеседник тихо рассмеялся. — Только это не радио.

Меня срочно доставили на корпоративном реактивном самолете в научно-исследовательский центр одной большой компании, расположенный где-то к востоку от штата Калифорния. Они были настолько озабочены сохранением секретности, что мне так никогда и не пришлось увидеть разрабатываемое ими устройство. Тем не менее они отличнейшим образом абстрагировали задачу.

Задача была связана с технологией производства, получившей название *селективная сборка*. Эли Уитни (Eli Whitney) считается изобретателем системы *взаимозаменяемых компонентов*, которая сыграла важную роль в промышленной революции. Принцип взаимозаменяемости состоит в изготовлении деталей механизма с определенной степенью точности, называемой *допуском на обработку*, что позволяет свободно заменять при сборке механизма одни подходящие детали на другие. Это существенно ускоряет процесс производства, поскольку избавляет сборщиков от необходимости индивидуально подгонять каждую деталь с помощью напильника. Взаимозаменяемость также значительно облегчила ремонт изделий, позволяя выполнять замену вышедших из строя деталей без особых трудностей. И все было бы очень хорошо, если бы не одно обстоятельство.

В частности, если размеры детали слегка отклонялись от установленных допусков, такую деталь включить в сборку было уже нельзя. Однако нашелся сообразительный сотрудник, который предложил использовать детали с отклонениями в допусках, взаимно компенсирующими друг друга. Детали, изготовленные с размерами «в минус», вполне могли быть сопряжены с деталями, изготовленными с размерами «в плюс», что давало приемлемый результат. В этом и заключается суть *селективной сборки*.

— Каждое устройство состоит из $n$ деталей разных типов, — продолжал представитель компании. — Для $i$-го типа детали (скажем, фланцевой прокладки) имеется $s_i$ ее экземпляров, и по каждому из них указана степень отклонения от эталонного размера. Нам необходимо подобрать детали друг к другу таким образом, чтобы получить максимально возможное количество работающих приборов.

— Предположим, устройство состоит из трех деталей, а сумма отклонений от нормы всех деталей работающего устройства не должна превышать 50. Умело подбирая для каждого устройства детали «в плюс» с деталями «в минус», мы можем использовать все детали и получить три работающих устройства. Эта ситуация показана на рис. 12.11.

Я немного поразмышлял над задачей. Проще всего было бы использовать для сборки каждого устройства самые лучшие оставшиеся детали каждого типа, повторяя процесс до тех пор, пока собранное устройство не будет работать. Но так мы получим небольшое количество устройств с большим разбросом качества, в то время как компании требуется максимальное количество устройств высокого качества.

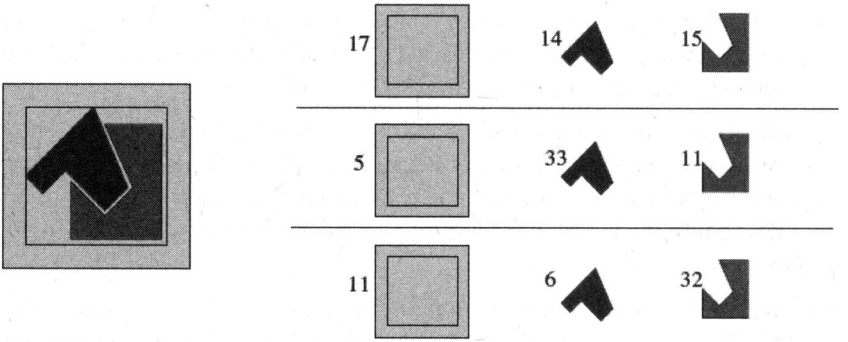



**Рис. 12.11.** Распределение деталей между тремя устройствами с тем, чтобы сумма отклонений для каждого не превышала 50

Целью было совместить детали «в плюс» с деталями «в минус» друг с другом таким образом, чтобы общее качество собранного устройства оставалось приемлемым. Определенно, задача выглядела как *паросочетание* в графах (см. *разд. 18.6*). Допустим, мы создадим граф, в котором вершины представляют экземпляры деталей, и соединим ребрами все пары экземпляров с допустимой общей погрешностью в размерах. В паросочетании в графах мы ищем наибольшее количество ребер, в котором никакие два ребра не опираются на одну и ту же вершину. Это аналогично максимальному количеству сборок из двух деталей, которые можно получить из имеющегося набора деталей.

— Вашу задачу можно решить методом паросочетания в двудольном графе, — объявил я. — При условии, что все устройства состоят из двух деталей.

Мое заявление было встречено всеобщим молчанием, за которым последовал дружный смех: «*Все* знают, что в устройстве больше двух деталей».

Так что предложенный мною алгоритмический подход был отвергнут. Расширение же задачи до сопоставления более чем двух деталей превращало ее в задачу паросочетания в гиперграфах[4], которая является NP-полной. Более того, только само время построения графа может быть экспоненциально зависимым от количества типов деталей, поскольку каждое возможное гиперребро (сборку) нужно будет создавать явным образом.

Пришлось начинать разработку алгоритма сначала. Итак, нужно собрать детали таким образом, чтобы общая сумма отклонений от нормы не превышала определенного допустимого значения. Это выглядело как *задача упаковки контейнеров*. В задаче такого типа (см. *разд. 20.9*) требуется разложить набор элементов разного размера в наименьшее количество контейнеров с ограниченной емкостью $k$. В нашем случае контейнеры представляли сборки, каждая из которых допускала общую сумму отклонений от нормы $\leq k$. Пакуемыми элементами были отдельные детали, размер которых отражал качество изготовления.

Однако полученная задача не является чистой задачей разложения по контейнерам в силу неоднотипности деталей и накладывает ограничение на допустимое содержимое

[4] Гиперграф содержит ребра, каждое из которых может иметь более двух вершин. Их можно представить в виде общих коллекций подмножеств вершин/элементов.

контейнеров. Требование создать наибольшее количество устройств означало, что нужно найти метод разложения элементов, который максимизирует количество контейнеров, содержащих ровно одну деталь каждого типа.

Задача разложения по контейнерам является NP-полной, однако естественно подходит для решения методом эвристического поиска. Пространство решений состоит из вариантов разложения деталей по контейнерам. Каждому контейнеру мы сначала присваиваем по произвольной детали каждого типа, чтобы обеспечить начальную конфигурацию для поиска.

Потом мы выполняем локальный поиск, перемещая детали из одного контейнера в другой. Можно было бы перемещать по одной детали, но более эффективно *обменивать* детали одинакового типа между двумя произвольно выбранными контейнерами. При таком обмене оба контейнера остаются полностью собранными устройствами с предположительно более приемлемым значением общего отклонения от нормы. Для операции обмена требовалось три произвольных целых числа: одно — для выбора типа детали (в диапазоне от 1 до $m$) и два — для выбора контейнеров, между которыми нужно выполнять обмен (в диапазоне от 1 до $b$).

Ключевым решением стал выбор функции стоимости. Для каждой сборки был установлен жесткий общий предел отклонений от нормы, равный $k$. Но что может быть наилучшим способом оценки *нескольких* сборок? Можно было бы в качестве общей оценки просто возвращать количество приемлемых сборок — целое число в диапазоне от 1 до $b$. Хотя эта величина и оставалась той, которую мы хотели оптимизировать, она не была достаточно чувствительна к продвижению в направлении правильного решения. Допустим, что в результате одного из обменов деталей между сборками допустимое отклонение одной из неработающих сборок стало намного ближе к пределу отклонений $k$ для сборки. Это стало бы значительно более удачной отправной точкой для дальнейшего поиска решения, чем первоначальная.

В итоге я остановился на следующей функции стоимости. Каждой работающей сборке я давал оценку в 1 пункт, а каждой неработающей — значительно меньшую оценку, зависящую от того, насколько близка она была к предельному значению $k$. Оценка неработающей сборки понижалась экспоненциально — в зависимости от того, насколько ее общее отклонение от нормы превышало $k$. Таким образом, оптимизатор станет пытаться максимизировать количество работающих устройств, после чего будет стараться понизить количество дефектов в другой сборке, которая была близкой к пределу.

Я воплотил этот алгоритм в коде, после чего обработал с его помощью набор деталей, взятых непосредственно из производственного цеха. Оказалось, что изготавливаемые устройства содержат по 8 деталей важных типов. Детали некоторых типов были более дорогими, чем другие, поэтому из них было доступно меньше кандидатов для рассмотрения. Деталей с наименьшими разрешенными отклонениями имелось только восемь экземпляров, поэтому из представленного набора деталей можно было собрать только восемь устройств.

Я наблюдал за работой программы имитации отжига. Первые четыре сборки она выдала сразу же без особых усилий, и лишь после этого поиск немного замедлился, и сборки 5 и 6 были предоставлены с небольшой задержкой. Потом наступила пауза, после которой была выдана седьмая сборка. Но, несмотря на все свои усилия, программа не

смогла скомпоновать восемь работающих устройств, прежде чем мне надоело наблюдать за ее действиями.

Я позвонил в компанию и хотел признаться в поражении, но они и слышать ничего об этом не хотели. Оказалось, что самостоятельно они смогли получить только шесть работающих устройств, так что мой результат стал для них большим успехом.

## 12.8. История из жизни. Отжиг массивов

В истории из жизни, приведенной в *разд. 3.9*, рассказывалось, как мы использовали структуры данных высшего уровня для разработки нового метода интерактивного секвенирования ДНК методом гибридизации. Для этого метода нужно было создавать по требованию массивы специфичных олигонуклеотидов.

Наш метод заинтересовал одного биохимика из Оксфордского университета и, что более важно, в его лаборатории было необходимое оборудование для испытания этого метода. Автоматическая система подачи реактивов Southern Array Maker (производства компании Beckman Instruments) выкладывает дискретные последовательности олигонуклеотидов в виде шестидесяти четырех параллельных рядов на полипропиленовую подложку. Устройство создает массивы, добавляя отдельные элементы в каждую ячейку вдоль определенных рядов и столбцов массива. На рис. 12.2 показан пример создания массива всех $2^4 = 16$ пуриновых ($A$ или $G$) 4-меров путем создания префиксов вдоль рядов и суффиксов вдоль столбцов.

| | Суффикс | | | |
|---|---|---|---|---|
| Префикс | *AA* | *AG* | *GA* | *GG* |
| *AA* | *AAAA* | *AAAG* | *AAGA* | *AAGG* |
| *AG* | *AGAA* | *AGAG* | *AGGA* | *AGGG* |
| *GA* | *GAAA* | *GAAG* | *GAGA* | *GAGG* |
| *GG* | *GGAA* | *GGAG* | *GGGA* | *GGGG* |

**Рис. 12.12.** Массив всех пуриновых 4-меров

Эта технология предоставляла идеальную среду для проверки возможности применения интерактивного секвенирования методом гибридизации в лаборатории, поскольку позволяла создавать с помощью компьютера широкий набор массивов олигонуклеотидов.

Ну а мы должны были предоставить соответствующее программное обеспечение. Для создания сложных массивов требовалось решить трудную комбинаторную задачу. В качестве входа задавался набор $n$ строк (представляющих олигонуклеотиды), из которых нужно было создать массив размером $m \times m$ (где $m = 64$) на устройстве Southern Array Maker. Нам надо было запрограммировать создание рядов и столбцов, чтобы реализовать множество строк $S$. Мы доказали, что задача разработки плотных массивов является NP-полной, но это не имело большого значения, поскольку решать ее в любом случае пришлось мне и моему студенту Рики Брэдли (Ricky Bradley).

— Нам придется использовать эвристический метод, — сказал я ему. — Как смоделировать эту задачу?

— Каждую строку можно разбить на составляющие ее префиксные и суффиксные пары. Например, строку *ACC* можно создать четырьмя разными способами: из префикса « » (пустая строка) и суффикса *ACC*, префикса *A* и суффикса *CC*, префикса *AC* и суффикса *C* или префикса *ACC* и суффикса « ». Нам нужно найти наименьший набор префиксов и суффиксов, которые совместно позволяют получить все такие строки, — сказал Рики.

— Хорошо. Это дает нам естественное представление для метода имитации отжига. Пространство состояний будет содержать все возможные подмножества префиксов и суффиксов. Естественные переходы между состояниями могут включать вставку или удаление строк из наших подмножеств или обмен пар местами.

— Какую функцию стоимости можно использовать? — спросил он.

— Нам требуется массив как можно меньшего размера, который охватывает все строки. Попробуем взять максимальное количество рядов (префиксов) или столбцов (суффиксов), используемых в нашем массиве, плюс те строки из множества *S*, которые еще не были задействованы.

Рики отправился на свое рабочее место и реализовал программу имитации отжига в соответствии с этими принципами. Программа выводила текущее состояние решения после каждого принятия перехода, и за ее работой было интересно наблюдать. Она быстро отсеяла ненужные префиксы и суффиксы, и размер массива стал стремительно сокращаться. Но после нескольких сотен итераций процесс начал замедляться. При переходе программа удаляла ненужный суффикс, выполняла вычисления в течение некоторого времени, а потом добавляла другой суффикс. После нескольких тысяч итераций не наблюдалось никакого реального улучшения.

— Кажется, что программа не может определить, когда она получает правильное решение, — предположил я. — Функция стоимости оценивает только минимизацию большей стороны массива. Давай добавим в нее выражения для оценки действий с другой стороной.

Рики внес соответствующие изменения в функцию стоимости, и мы снова запустили программу. На этот раз она уверенно обрабатывала и другую сторону. Более того, наши массивы стали выглядеть как тонкие прямоугольники, а не квадраты.

— Ладно. Давай добавим в функцию стоимости еще одно выражение, чтобы массивы стали квадратными.

Рики опять изменил программу. На этот раз массивы на выходе имели правильную форму, и поиск двигался в нужном направлении. Но происходило это по-прежнему медленно.

— Многие операции вставки затрагивают недостаточное количество строк. Может быть, нам следует сделать уклон в пользу случайной выборки, чтобы важные префиксы и суффиксы выбирались чаще?

Рики внес очередные изменения. Теперь программа работала быстрее, но все равно иногда возникали паузы. Мы изменили график охлаждения. Результаты улучшились,

но были ли они действительно хорошими? Не зная нижнего оптимального предела, нельзя было сказать, насколько удачным является наше решение. Мы продолжали настраивать разные аспекты нашей программы, пока все дальнейшие модификации не перестали приводить к улучшению.

Конечная версия программы улучшала первоначальный массив, используя следующие операции:

- `swap` — обменять префикс/суффикс в массиве с префиксом/суффиксом вне массива;
- `add` — добавить в массив случайный префикс/суффикс;
- `delete` — удалить из массива случайный префикс/суффикс;
- `useful add` — добавить в массив префикс/суффикс с наибольшей пригодностью;
- `useful delete` — удалить из массива префикс/суффикс с наименьшей пригодностью;
- `string add` — выбрать произвольную строку вне массива и добавить наиболее пригодный префикс и/или суффикс, чтобы покрыть эту строку.

Мы использовали стандартный график охлаждения с экспоненциально убывающей температурой (зависящей от размера задачи) и зависящим от температуры критерием Больцмана, чтобы выбирать состояния, имеющие более высокую стоимость. Наша конечная функция стоимости определялась следующим образом:

$$cost = 2 \times max + min + \frac{(max - min)^2}{4} + 4(str_{total} - str_{in}),$$

где $max$ — максимальный размер чипа, $min$ — минимальный его размер, $str_{total} = |S|$, а $str_{in}$ — количество строк из множества $S$, находящихся в настоящее время в чипе.

Насколько хороший результат мы получили? На рис. 12.13 показано четыре этапа сжатия массива, состоящего из 5716 однозначных 7-меров вируса иммунодефицита человека (ВИЧ), — после 0, 500, 1000 и, наконец, 5750 итераций.



**Рис. 12.13.** Сжатие массива ВИЧ методом имитации отжига после 0, 500, 1000 и 5750 итераций

Пикселы массива представляют здесь первое появление 7-мера ВИЧ. Конечный размер чипа — 130×132, что явно лучше по сравнению с первоначальным размером 192×192. Чтобы завершить оптимизацию, программе потребовалось около 15 минут, и это вполне приемлемо для решаемой задачи.

Насколько хорошим был полученный результат? Так как имитация отжига — всего лишь эвристический метод, то мы, по сути, не знаем, насколько близким к оптималь-

ному является наше решение. Лично я думаю, что результат достаточно хороший, но не могу быть в этом полностью уверенным. Метод имитации отжига является хорошим инструментом для решения сложных задач оптимизации. Однако, если вы хотите получить отличные результаты, будьте готовы потратить больше времени на дополнительную настройку и оптимизацию программы, чем ушло у нас на создание ее первоначального варианта. Это тяжелая работа, но зачастую иного выхода нет.

## 12.9. Генетические алгоритмы и другие эвристические методы поиска

Для решения задач комбинаторной оптимизации предложено много эвристических методов. Подобно методу имитации отжига, многие из этих методов основаны на аналогии с реальными физическими процессами, включая *генетические алгоритмы*, *нейронные сети* и *алгоритм муравейника*.

Интуитивные представления, лежащие в основе этих методов, привлекательны своей понятностью, но скептики считают такие подходы шаманством, основанным на поверхностных аналогиях с природными явлениями, а не средством получения лучших по сравнению с другими методами вычислительных результатов для реальных задач.

Вопрос заключается не в том, возможно ли, приложив достаточные усилия, получить приемлемые решения задач с помощью этих методов. Ясно, что возможно. Но настоящий вопрос состоит в том, позволяют ли эти методы получить *лучшие* решения при *меньшей сложности реализации* или *более высокой эффективности*, чем другие рассмотренные методы.

В общем, я так не думаю. Но в духе непредвзятого исследования мы кратко рассмотрим генетические алгоритмы, которые пользуются наибольшей популярностью. Более подробную информацию о них вы найдете в *разд. «Замечания к главе»*.

### 12.9.1. Генетические алгоритмы

Идея генетических алгоритмов возникла как отражение теории эволюции и естественного отбора. Посредством естественного отбора организмы приспосабливаются к выживанию в определенной среде. В генетическом коде организма происходят случайные мутации, которые передаются его потомкам. Если та или иная мутация окажется полезной, то тогда у потомков будет больше шансов выжить. Если же мутация вредная, то вероятность выживания потомков и передачи этого качества по наследству уменьшается.

Генетические алгоритмы содержат «популяцию» кандидатов на решение этой задачи. Из этой популяции случайным образом выбираются элементы, которые «размножаются» так, что в потомке комбинируются свойства двух родителей. Вероятность выбора элемента для размножения основывается на его «физической форме», которой, по существу, является качество предоставляемого им решения. Элементы, имеющие плохую физическую форму, удаляются из популяции, уступая место потомкам элементов, пребывающих в более хорошей форме.

Идея генетических алгоритмов чрезвычайно привлекательна. Но эти алгоритмы просто не могут работать столь же эффективно над практическими задачами комбинаторной оптимизации, как метод имитации отжига. На то есть две причины. Во-первых, моделирование приложений посредством генетических операторов, таких как мутация и пересечение на битовых строках, является в высшей степени неестественным. Такая псевдобиология вносит в решаемую задачу дополнительный уровень сложности. Во-вторых, решение нетривиальных задач посредством генетических алгоритмов занимает очень много времени. Операции пересечения и мутации обычно не используют структуры данных, специфичных для той или иной задачи, вследствие чего большинство переходов дают низкокачественные решения и процесс поиска наилучшего решения протекает медленно. В этом случае аналогия с эволюцией (которой для осуществления важных изменений требуются миллионы лет) оказывается уместной.

Мы не будем продолжать обсуждение генетических алгоритмов — лишь попытаемся отговорить вас от их использования в своих приложениях. Но если вы все же очень хотите поэкспериментировать с ними, то подсказки по их реализации можно найти в *разд. 16.5*.

#### Подведение итогов

Я лично *никогда* не сталкивался ни с одной задачей, для решения которой генетические алгоритмы оказались бы самым подходящим средством. Более того, я никогда не встречал никаких результатов вычислений, полученных посредством генетических алгоритмов, которые бы производили на меня положительное впечатление. Пользуйтесь методом имитации отжига для своих экспериментов с эвристическим поиском.

## 12.10. Квантовые вычисления

Мы живём в эпоху, когда модель вычислений RAM (Random Access Machine, машина с произвольным доступом), с которой мы познакомились в *разд. 2.1*, дополняется новым классом вычислительных устройств. Эти устройства работают на принципах квантовой механики, которая приписывает, казалось бы, невозможные свойства тому, как должны быть организованы системы атомов.

Квантовые компьютеры используют эти свойства для выполнения определенных типов вычислений с более высокой асимптотической эффективностью алгоритмов, чем возможно на обычных компьютерах. Хорошо известно, что квантовая механика настолько трудно поддается интуитивному пониманию, что никто по настоящему ее не понимает. Суперпозиция! Квантовая странность! Запутанность! Кот Шрёдингера! Функции спадающей волны! Аааа!!! Я должен пояснить, что нет абсолютно никаких разногласий о правилах работы квантовых компьютеров. Знающие люди сходятся во мнении о свойствах квантовой механики и теоретических возможностях этих машин. Не обязательно понимать закон физики, чтобы беспрекословно его выполнять. Исследования в области квантовых вычислений направлены на разработку методов реализации больших и надежных квантовых систем и создание новых алгоритмов для использования этих возможностей.

Я полагаю, что большинство читателей этой книги, скорее всего, никогда не изучали «высокую» физику, а ваши знания линейной алгебры и комплексных чисел притупи-

лись, поэтому я постараюсь не затрагивать эти области. Моя цель состоит в том, чтобы показать, почему квантовые компьютеры обладают большими потенциальными возможностями, а также хотя бы немного разъяснить, как можно использовать их возможности, чтобы получить алгоритмы с более высокой асимптотической эффективностью для решения определенных задач. Для этого я применю подход с созданием новой модели «квантового» компьютера. Моя модель не совсем правильная, но, я надеюсь, она позволит понять, почему эти машины вызывают такое восхищение. Далее мы вкратце рассмотрим работу трех наиболее известных квантовых алгоритмов. Наконец, я дам несколько предсказаний касательно будущего квантовых вычислений и укажу на неточности в моей модели (см. *разд. 12.10.5*).

## 12.10.1. Свойства «квантовых» компьютеров

Рассмотрим обычный детерминированный компьютер с $n$ битами памяти, обозначенными $b_0$, ..., $b_{n-1}$. Поскольку каждый из этих битов может иметь значение или 0 или 1, то наш компьютер может представлять ровно $N = 2^n$ состояний. Состояние номер $i$ машины определяется строкой битов, соответствующих двоичному представлению целого числа $i$. Каждая исполняемая команда изменяет состояние машины, меняя состояние определенного множества битов на обратное.

Можно считать, что обычный детерминированный компьютер поддерживает распределение вероятностей своего текущего состояния. В любой момент времени вероятность $p(i)$ нахождения в состоянии $i$ равна 0 для $2^n - 1$ изо всех возможных состояний, а $p(j) = 1$, если компьютер находится в состоянии $j$. Да, это распределение вероятности по состояниям, но не очень интересное.

Квантовый компьютер оснащается $n$ *кубитами* (qubit, квантовый бит) памяти, обозначаемыми $q_0$, ..., $q_{n-1}$. Для такого компьютера также существует $N = 2^n$ возможных комбинаций битов, но фактическое состояние компьютера в любой момент является функцией распределения вероятности. С каждой из $2^n$ битовых комбинаций связана вероятность $p(i)$ того, что при считывании состояния машины это состояние будет $i$. Такое распределение вероятностей намного богаче, чем для обычных детерминированных машин, — в любое время возможна ненулевая вероятность нахождения машины во всех $N$ состояниях. Возможность манипулирования распределением вероятностей одновременно для всех $N = 2^n$ состояний и является настоящим достоинством квантовых вычислений. Как и в случае с любым распределением вероятностей, сумма всех этих вероятностей должна составлять 1, так что

$$\sum_{i=0}^{N-1} p(i) = 1.$$

«Квантовые» компьютеры поддерживают следующие операции:

- `Initialize-state(Q,n,D)` (Инициализация состояний).

  Инициализирует распределение вероятностей $n$ кубитов машины $Q$ согласно описанию $D$. Очевидно, что если бы параметр $D$ был задан в виде явного списка требуемых вероятностей для каждого состояния, то эта операция заняла бы время $\Theta(2^n)$. Поэтому мы стремимся использовать общие описания небольшого — скажем, $\Theta(n)$ — размера, например: «задать всем $N = 2^n$ состояниям одинаковую вероят-

ность, чтобы $p(i) = 1/2^n$». Время выполнения операции `Initialize-state` составляет $O(|D|)$, а не $O(N)$.

- `Quantum-gate(Q,c)` (Квантовый вентиль).

  Изменяет распределение вероятностей машины $Q$ согласно условию квантового вентиля $c$. Квантовые вентили выполняют логические операции наподобие операций И или ИЛИ, изменяя вероятности состояний согласно текущему содержимому кубитов, — скажем, $q_x$ и $q_y$. Время исполнения этой операции пропорционально количеству кубитов, задействованных с состоянием $c$, но обычно составляет $O(1)$.

- `Jack(Q,c)` (Повысить).

  Повышает вероятности всех состояний, определяемых условием $c$. Например, мы хотим повысить вероятности всех состояний, где условие $c$ = «кубит $q_2 = 1$», как показано на рис. 12.14. Время исполнения этой операции пропорционально количеству кубитов в состоянии $c$, но обычно составляет $O(1)$.




**Рис. 12.14.** Повышение вероятности всех состояний, где кубит $q_2 = 1$

То, что эту операцию можно выполнить за константное время, следует понимать как не совсем заурядное обстоятельство. Даже если условие повышает вероятность всего лишь одного состояния $i$, то для того, чтобы сумма всех вероятностей оставалась равной 1, вероятности всех остальных $2n - 1$ состояний необходимо понизить. И то, что это можно выполнить за константное время, является одним из странных свойств «квантовой» физики.

- `Sample(Q)` (Выборка).

  Произвольно выбирает ровно одно из $2^n$ состояний согласно текущему распределению вероятностей машины $Q$ и возвращает значения $n$ кубитов $q_0, ..., q_{n-1}$. Эта операция занимает $O(n)$ время.

«Квантовые» алгоритмы представляют собой последовательности этих операций, возможно, дополненных управляющей логикой от обычных компьютеров.

## 12.10.2. Алгоритм Гровера для поиска в базе данных

Первым мы рассмотрим алгоритм для поиска в базе данных, или, в более широком смысле, инверсию функций. Предположим, что нас интересует однозначная двоичная строка, определяющая каждое из $N = 2^n$ состояний как ключ, где каждому состоянию $i$ соответствует значение $v(i)$. Если для хранения этих значений требуется $m$ битов, то нашу базу данных можно рассматривать как состоящую из $2^n$ неупорядоченных строк, с ненулевой вероятностью, что длина каждой из них составляет $n + m$ кубитов со значением $v(i)$, хранящимся в $m$ старших кубитах.

Такую систему $Q$ можно создать, используя команду `Initialize-state(Q,n,D)` с соответствующим условием $D$. Любой строке вида $i$ длиной в $n + m$ кубитов и конкатенированной со значением $v(i)$, присвоим вероятность величиной $1/2^n$. Всем остальным из $2^m - 1$ строк длиной $n + m$ кубитов с префиксом $i$ присвоим вероятность величиной 0. Для строки поиска $S$ длиной $m$ битов мы хотим возвратить такую строку длиной $n + m$ кубитов, чтобы $S = q_n \ldots q_{n+m-1}$.

Представленный в предыдущем разделе набор команд не содержит команды вывода, кроме команды `Sample(Q)`. Чтобы сделать вероятным возвращение этой выборкой требуемых нам данных, нужно повысить вероятность всех строк с правильными значениями кубитов. Предлагаемый алгоритм для реализации всего этого приводится в листинге 12.9.

**Листинг 12.9. Алгоритм для поиска в базе данных**

```
Search(Q,S)
    Repeat
        Jack(Q, "все строки, в которых S = qn . . . qn+m-1")
    Пока вероятность успеха достаточно высока
    Возвращаем первые n битов из  Sample(Q)
```

Каждая операция `Jack` выполняется с высокой скоростью — за константное время. Но в целом она повышает вероятности настолько медленно, что для успешного поиска необходимо и достаточно выполнить $\Theta\left(\sqrt{N}\right)$ циклов. То есть этот алгоритм возвращает соответствующую строку за время $\Theta\left(\sqrt{N}\right)$, что значительно лучше, чем временная сложность последовательного поиска величиной $\Theta(N)$.

### Решение задачи о выполнимости

Алгоритму Гровера для поиска в базе данных можно найти интересное применение — решение задачи о выполнимости (см. *разд. 11.4*), которая является самой трудной из NP-сложных задач. Как указывалось ранее, квантовая система с $n$ кубитами представляет все $N = 2^n$ двоичные строки длиной $n$. Рассматривая значение 1 как *истина*, а 0 — как *ложь*, каждая такая строка определяет набор значений истинности для $n$ булевых переменных.

Теперь добавим в систему $(n + 1)$-й кубит для хранения флага, указывающего, удовлетворяет ли $i$-я строка логической булевой формуле $F$. Тестирование, удовлетворяет ли тот или иной набор значений истинности какому-либо множеству дизъюнкций, не со-

ставляет никакого труда — просто явно проверяем, содержит ли каждая дизъюнкция, по крайней мере, один литерал со значением *истина*. Эту проверку можно выполнять одновременно, используя последовательность квантовых вентилей. Для 3-SAT формулы $F$, содержащей $k$ дизъюнкций, такую проверку можно выполнить, сканируя каждую дизъюнкцию и используя, в общем, приблизительно $3k$ квантовых вентилей. Если $q_0$, ..., $q_{n-1}$ является выполняющим набором значений истинности, кубиту $q_n$ присваивается значение 1 — или 0 в противном случае. Это означает, что всем строкам $s$, соответствующим неправильным значениям $F$, присваивается значение $p(s) = 0$.

Теперь предположим, что мы выполним операцию `Search(Q,(`$q_n$ `== 1))`, которая возвратит квантовое состояние $\{q_0, ..., q_{n-1}\}$. Существует высокая вероятность, что это будет выполняющий набор значений истинности, и, таким образом, мы получаем эффективный алгоритм для решения задачи выполнимости.

Время исполнения поискового алгоритма Гровера составляет $O\left(\sqrt{N}\right)$, где $N = 2^n$. Так как $\sqrt{N} = \sqrt{2^n} = \left(\sqrt{2}\right)^n$, то время исполнения равно $O\left(1{,}414^n\right)$, что, по сравнению с примитивной границей, является большим улучшением. Для $n = 100$ это уменьшает количество шагов с $1{,}27 \cdot 10^{30}$ до $1{,}13 \cdot 10^{15}$. Но тем не менее $\left(\sqrt{2}\right)^n$ возрастает экспоненциально, поэтому это не алгоритм с полиномиальным временем исполнения. То есть рассмотренный нами квантовый алгоритм представляет большое улучшение по сравнению с поиском полным перебором, но недостаточно большое, чтобы вызвать эффект $P = NP$.

#### Подведение итогов

Несмотря на их мощь, *квантовые компьютеры не способны решать NP-полные задачи за полиномиальное время*. Конечно же, если $P = NP$, мир меняется, но, предположительно, $P \neq NP$. Мы полагаем, что класс задач, которые можно решить за полиномиальное время на квантовом компьютере (они называются BQP-задачами), не содержит NP-задач, — примерно с такой же уверенностью, как и то, что P ≠ NP.

## 12.10.3. Более быстрое «преобразование Фурье»

Наиболее важный алгоритм в области обработки сигналов — быстрое преобразование Фурье (БПФ) — преобразовывает числовой временной ряд из $N$ элементов в эквивалентное представление в виде суммы $N$ периодических функций разных частот. Многие алгоритмы фильтрации и сжатия сигналов понижают высокие и/или низкие частотные составляющие сигнала до их полного устранения из преобразования Фурье, как рассматривается в *разд. 16.11*.

В большинстве случаев для вычисления преобразований Фурье используются алгоритмы с временем исполнения $O(N^2)$, где каждый из $N$ элементов создается в виде суммы $N$ членов. Быстрое преобразование Фурье — это алгоритм типа «разделяй и властвуй» для вычисления этой свертки за время $O(N \log N)$, как показано в *разд. 5.9*. Алгоритм БПФ можно также реализовать в виде схемы с $\log M$ этапами, где на каждом этапе выполняется $M$ независимых параллельных операций умножения.

Таким образом, получается, что каждый из этих этапов схемы БПФ можно реализовать, используя $\log M$ квантовых вентилей. То есть преобразование Фурье для $N = 2^n$

состояний $n$-кубитной системы можно вычислить на «квантовом» компьютере за время $O((\log N)^2) = O(n^2)$. А это экспоненциальное улучшение по сравнению с БПФ! Но не все так просто. Сейчас мы имеем $n$-кубитовую квантовую систему $Q$, в которой коэффициент Фурье, связанный с каждым входным элементом $0 \leq a_i \leq N-1$, представлен вероятностью строки $i$ в $Q$. Выполнить вывод этих $2^n$ коэффициентов из машины $Q$ абсолютно невозможно. Все, что мы можем сделать, это вызвать команду `Sample(Q)` и получить индекс (предположительно) большого коэффициента, выбранный с вероятностью, пропорциональной его величине.

Это «преобразование Фурье» очень ограниченного типа, и оно возвращает только индекс ячейки памяти, которая с высокой вероятностью может содержать большой коэффициент. Но оно подготавливает почву для, наверное, самого знаменитого квантового алгоритма — алгоритма Шора для разложения целых чисел на множители.

## 12.10.4. Алгоритм Шора для разложения целых чисел на множители

Существует интересная связь между периодическими функциями (т. е. функциями, повторяющими свои значения через некоторый регулярный интервал аргумента) и целыми числами, кратными какому-либо целому числу $k$ или, что эквивалентно, делящимися на него. Должно быть ясно, что на числовой прямой эти целые числа должны находиться на расстоянии $k$ друг от друга. Какие целые числа кратны числу 7? Это числа 7, 14, 21, 28 и т. д. — что есть явно периодическая функция с периодом $k = 7$.

Быстрое преобразование Фурье позволяет нам преобразовать ряд во «временной области» (который в рассматриваемом случае состоит из всех чисел, кратных 7 и меньших чем $N$) в «частотную область» с ненулевым значением для седьмого коэффициента Фурье, означающим повторяющуюся функцию с периодом 7, и наоборот. На рис. 12.15 показана $n$-кубитовая квантовая система $Q$, каждое состояние $i$ которой инициализировано вероятностью $p(i) = 1/2^n$, представляющей период каждого возможного множителя $i$. Быстрое преобразование Фурье над $Q$ переводит нас во временную область, создавая несколько множителей для каждого целого числа $0 \leq i \leq N-1$.

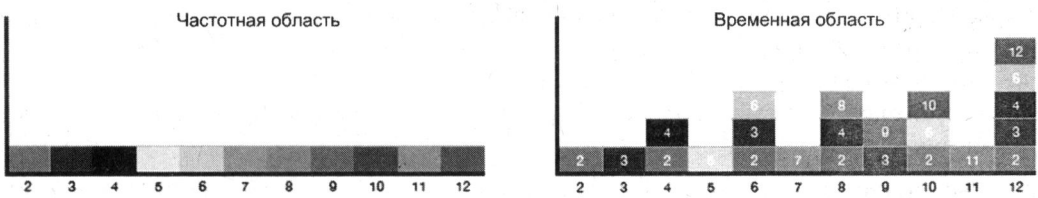



**Рис. 12.15.** Преобразование целых чисел из частотной области во временную вычисляет количество множителей для каждого целого числа

Теперь предположим, что мы хотим разложить на множители какое-либо целое число $M < N$. Используя квантовое волшебство, можно организовать систему $Q$ во временной области, где только целые числа, кратные множителям $M$, имеют высокие вероятности (рис. 12.16).

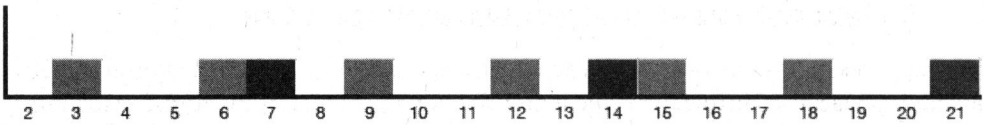



**Рис. 12.16.** Маловероятно, что множитель можно получить напрямую выборкой из чисел, кратных множителям значения $M$ (здесь 21), но он почти наверняка будет получен из наибольшего общего делителя

Каждая выборка из этой системы дает нам произвольное число, кратное множителю значения $M$. Например, для $M = 77 = 7 \times 11$ можно получить значения выборок 33, 42 и 55. Эти значения могут выглядеть обещающими, но на поверку ни одно из них не является множителем значения $M$.

*Наибольший общий делитель* нод($x$, $y$) — это наибольшее целое число $d$, на которое делится как число $x$, так и число $y$. Для вычисления нод($x$, $y$) существуют быстрые алгоритмы — некоторые из них мы рассмотрели в *разд. 11.2.3*. Любой наибольший общий делитель больше чем 1 будет отличным кандидатом для множителя значения $M$. Обратите внимание на то, что нод(33, 55) = 11.

В листинге 12.10 приводится псевдокод полного алгоритма Шора для разложения целых чисел на множители.

**Листинг 12.10. Псевдокод алгоритма Шора для разложения целых чисел на множители**

```
Factor(M)
    Организовываем n-кубитовую квантовую систему Q, где N = 2^n и M < N.
    Инициализируем Q, чтобы p(i) = 1/2^n для всех 0 ≤ i ≤ N - 1.
    Повторяем
        Jack(Q, "все i, для каких (нод(i,M) > 1)")
    до тех пор, пока вероятности всех членов, взаимно простых к числу M,
    остаются очень малыми.
    FFT(Q).
    For j = 1 to n
        S_j = Sample(Q)
        Если ((d = GCD(S_j, S_k)) > 1) и (d делит M)), для некоего k < j
            Возвращаем (d) как множитель для M
        В противном случае докладываем, что множитель не найден
```

Время выполнения каждой из этих операций пропорционально $n$, а не $M = \Theta(2^n)$, т. е. это повышение эффективности до экспоненциального времени по сравнению с разложением на множители методом деления и проверки. Неизвестно о существовании алгоритмов с полиномиальным временем исполнения для разложения целых чисел на множители на обычных компьютерах. Таким образом, существование быстрого алгоритма для разложения целых чисел на множители не нарушает никаких допущений теории сложности.

## 12.10.5. Перспективы квантовых вычислений

Каковы перспективы квантовых вычислений? Во время работы над этой книгой в 2020 году развитие в области квантовых вычислений происходит в быстром темпе. Мое ви́дение на этот счет не обязательно лучше, чем у других, но я сделаю несколько обоснованных предположений:

- *Квантовые вычисления — это стоящая вещь, и они будут воплощены в жизнь.*

  Я, как человек, который в течение сорока лет наблюдает за циклами ажиотажа в технологиях, развил достаточно надежный нюх на всякий отстой, и в настоящее время квантовые вычисления проходят мою проверку на вшивость с большим запасом. Я вижу очень умных людей, взбудораженных перспективами в этой области, очевидный и постоянный технологический прогресс, а также значительные инвестиции в нее крупными компаниями и другими важными игроками. И я бы очень удивился, если бы все это полностью сошло на нет.

- *Квантовые вычисления вряд ли окажут воздействие на рассматриваемые в этой книге задачи.*

  Специальные знания, полученные в результате прочтения этой книги, будут иметь ту же ценность и в эпоху квантовых вычислений. Я отношу их к технологиям для специализированных применений — наподобие того, как самые быстрые суперкомпьютеры в основном используются для научных вычислений, а не для решения задач промышленности.

  За исключением задачи разложения целых чисел на множители, я не вижу никаких других из рассматриваемых в этой книге задач, которые можно было бы потенциально лучше решить на квантовом компьютере. Самая быстрая технология не обязательно завоевывает мир. Например, для реализации наивысшей скорости передачи данных можно использовать громадный самолет, набитый DVD-дисками или носителями данных с еще большей плотностью записи. Тем не менее никто еще не придумал способа коммерческого применения этой технологии. Подобным образом, квантовые вычисления не обязательно хорошо впишутся в большинство обычных вычислений.

- *Большие достижения, скорее всего, будут связаны с задачами, которым специалисты в компьютерной области не придают особого значения.*

  Пока еще не ясно, каким будет основное применение квантовых вычислений, но похоже, что наиболее обещающие применения связаны с эмулированием квантовых систем. Это очень важно для химии и материаловедения и может проложить путь к поразительным революционным открытиям в разработке новых лекарств и инженерии. Но насколько велика будет роль специалистов в компьютерной области в проторении этого пути, пока неясно.

Поживем — увидим. Я с нетерпением ожидаю возможности выпустить четвертое издание этой книги, возможно, в 2035 году, чтобы узнать, насколько точными были мои предсказания.

Следует иметь в виду, что между описанными здесь «квантовыми» вычислениями и настоящими квантовыми компьютерами существует несколько важных различий, хотя

я считаю, что мое описание раскрывает суть их работы. Тем не менее в настоящих квантовых компьютерах:

- *Роль вероятностей играют комплексные числа.*

  Вероятности — это *действительные* числа в диапазоне от 0 до < 1, сумма которых для всех элементов вероятностного пространства составляет 1. А квантовые вероятности — это *комплексные* числа, квадраты которых находятся в диапазоне от 0 до < 1, и сумма которых для всех элементов вероятностного пространства составляет 1. Вспомним, что рассмотренный в *разд. 5.9* алгоритм быстрого преобразования Фурье применим к комплексным числам, что и является основой его мощи.

- *Считывание состояния квантовой системы уничтожает его.*

  При взятии выборки одного произвольного состояния квантовой системы вся информация об остальных $2^n - 1$ состояниях теряется. Таким образом, делать многократные выборки из распределения, как это делалось для нашей «квантовой» системы, невозможно. Но систему можно *воссоздавать* до начального состояния любое количество раз, и брать требуемые выборки из каждого из этих состояний, что создает эффект многократных выборок.

  Основное препятствие квантовых вычислений — вопрос, как извлечь требуемый ответ, поскольку это измерение дает только крошечную долю информации, заложенной в системе $Q$. Если измеряются только некоторые, а не все кубиты системы $Q$, тогда оставшиеся кубиты также «измеряются» — в том смысле, что их состояние соответственно разрушается. Это явление называется *запутанностью* (entanglement), и оно представляет собой настоящий источник волшебства квантовых вычислений.

- *Настоящие квантовые системы с легкостью испытывают сбои — декогерируют (decohere).*

  Манипулирование отдельными атомами для выполнения сложных задач — это не детская игра. Квантовые компьютеры — чтобы добиться от них как можно большей продолжительной работы — обычно работают при чрезвычайно низких температурах и в экранированной среде. При современном уровне развития технологий эта продолжительность не очень велика, что ограничивает степень сложности алгоритмов, которые можно исполнять на квантовых системах, и требует разработки технологий исправления ошибок для них.

- *Процесс инициализации квантовых состояний и способность квантовых вентилей несколько отличаются от того, как они описаны здесь.*

  Я позволил себе определенные вольности в описании того, как именно можно выполнять инициализацию квантовых состояний и какие операции они могут выполнять. Квантовые вентили, по сути, это унитарные матрицы, умножение на которые изменяет вероятности системы $Q$. Эти операции подробно определены свойствами квантовой механики, и такие детали весьма важны.

# Замечания к главе

Метод имитации отжига был первоначально рассмотрен в статье Киркпатрика (Kirkpatrick) и прочих [KGV83], где также обсуждалось его применение для решения задачи размещения на печатных платах сверхбольших интегральных схем. Приложения в *разд. 12.6.4* основаны на материале из книги Аартса (Aarts) и Корста (Korst) [AK89]. Компания D-Wave производит класс квантовых компьютеров, в которых предпринимаются попытки реализации квантового отжига для решения задач оптимизации, но пока еще неясно, насколько важна эта технология.

В представленных здесь эвристических решениях задачи коммивояжера в качестве локальной операции используется обмен вершин. В действительности намного более мощной операцией является обмен ребер. При каждой операции обмениваются максимум два ребра в маршруте, по сравнению с четырьмя ребрами при обмене местами вершин. Это повышает вероятность локального улучшения. Но чтобы эффективно поддерживать порядок полученного маршрута, требуются более сложные структуры данных, которые рассматриваются в книге Фредмана (Fredman) и прочих [FJMO93].

Разные эвристические методы поиска подробно представлены в книге Аартса (Aarts) и Ленстра (Lenstra) [AL97], которую я рекомендую всем, кто хочет углубить свои знания в области эвристических методов поиска. В этой книге описан алгоритм *поиска с запретами* (tabu search), который является разновидностью метода имитации отжига, использующей дополнительные структуры данных, чтобы избежать переходов в недавно рассмотренные состояния. Алгоритм муравейника рассматривается в книге Дориго (Dorigo) и Штутцла (Stutzle) [DT04]. В книге Ливната (Livnat) и Пападимитроу (Papadimitriou) [LP16] выдвигается теория, объясняющая повсеместную некачественность генетических алгоритмов: цель полового размножения заключается в создании разнородной популяции, а не высокооптимизированных отдельных личностей. Тем не менее Михалевич (Michalewicz) и Фогель (Fogel) в своей книге [MF00] излагают более доброжелательную точку зрения на генетические и им подобные алгоритмы, чем та, что представлена в этой главе.

Наша работа по использованию метода имитации отжига для сжатия массивов ДНК была представлена в журнале [BS97]. Дополнительную информацию по селективной сборке вы найдете в статье Пью (Pugh) [Pug86] и в статье Колларда (Coullard) и других [CGJ98].

Если мое введение в квантовые вычисления пробудило у вас интерес к ним, я бы настоятельно рекомендовал вам обратиться к более подробным источникам информации на эту тему. В частности, книги [Aar13, Ber19, DPV08] среди прочих предоставляют интересное рассмотрение квантовых вычислений, а в книге Яновского (Yanofsky) и Манучи (Mannucci) [YM08] этот материал излагается особенно понятно и доступно. Захватывающий блог Скотта Ааронсона (Scott Aaronson, **https://www.scottaaronson.com/blog/**) представляет последние достижения в алгоритмах квантовых вычислений, а также более широкий взгляд на теорию сложности вычислений.

# 12.11. Упражнения

## Частные случаи сложных задач

1. *[5]* Костяшки домино представляют пары целых чисел $(x_i, y_i)$, где каждое из значений $x_i$ и $y_i$ является целым числом в диапазоне от 1 до $n$. Пусть $S$ будет последовательностью из $m$ пар целых чисел $[(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)]$. Суть игры состоит в создании длинных цепочек $[(x_{i1}, y_{i1}), (x_{i2}, y_{i2}), ..., (x_{it}, y_{it})]$ таким образом, чтобы $y_{ij} = x_{i(j+1)}$. Костяшки можно разворачивать, т. е. $(x_i, y_i)$ эквивалентно $(y_i, x_i)$. Для последовательности $S = [(1, 3), (4, 2), (3, 5), (2, 3), (3, 8)]$ наиболее длинными цепочками среди прочих будут $[(4, 2), (2, 3), (3, 8)]$ и $[(1, 3), (3, 2), (2, 4)]$.

   a) Докажите, что задача вычисления самой длинной цепочки домино является NP-полной.

   b) Предоставьте эффективный алгоритм для вычисления самой длинной цепочки домино, в которой числа увеличиваются с увеличением длины цепочки. Например, для предыдущей последовательности $S$ такими цепочками будут $[(1, 3), (3, 5)]$ и $[(2, 3), (3, 5)]$.

2. *[5]* Даны граф $G = (V, E)$ и две различные вершины $x$ и $y$ в этом графе. За посещение каждой вершины $v$ выдается определенное количество жетонов $t(v)$.

   a) Докажите, что задача вычисления такого пути от $x$ до $y$, в котором можно получить наибольшее количество жетонов, является NP-полной.

   b) Предоставьте эффективный алгоритм для вычисления такого пути, если граф $G$ является бесконтурным ориентированным графом.

3. *[8]* В *задаче гамильтонова дополнения* требуется разработать алгоритм для добавления к графу $G$ такого наименьшего количества ребер, которое сделает его гамильтоновым графом. Для общих графов задача является NP-полной, но для ее решения с деревьями существует эффективный алгоритм. Разработайте эффективный и *доказуемо правильный* алгоритм для добавления в дерево $T$ минимально возможного количества ребер, чтобы сделать его гамильтоновым.

## Аппроксимирующие алгоритмы

4. *[4]* В *задаче максимальной выполнимости* требуется найти набор значений истинности, который является выполняющим для максимального количества дизъюнкций. Предоставьте эвристический алгоритм, выполняющий как минимум вдвое меньше дизъюнкций, чем это делает оптимальное решение.

5. *[5]* Имеется следующий эвристический алгоритм поиска вершинного покрытия: создается дерево обхода в глубину графа, из которого удаляются все листья, после чего оставшиеся узлы должны составлять вершинное покрытие графа. Докажите, что размер этого вершинного покрытия превышает оптимальный самое большее в два раза.

6. *[5]* В *задаче максимального разреза* графа $G = (V, E)$ требуется разделить множество вершин $V$ на два непересекающихся подмножества $A$ и $B$ таким образом, чтобы максимизировать количество ребер $(a, b) \in E$, где $a \in A$, $b \in B$. Рассмотрим следующий эвристический алгоритм поиска максимального разреза. Сначала помещаем вершину $v_1$ в подмножество $A$, а вершину $v_2$ — в подмножество $B$. Каждую оставшуюся вершину помещаем в подмножество, которое добавляет в разрез наибольшее количество ребер. Докажите, что размер этого разреза равен, по крайней мере, половине оптимального разреза.

7. *[5]* В *задаче об упаковке в контейнеры* дается $n$ элементов, веса которых равны $w_1$, $w_2$, ..., $w_n$. Нужно найти наименьшее количество контейнеров, в которые можно упаковать эти $n$ элементов, при этом емкость каждого контейнера не превышает один килограмм.

   В эвристическом алгоритме «первый подходящий» (first-fit) объекты рассматриваются в порядке, в котором они представляются. Каждый объект укладывается в первый же контейнер, в который он помещается. Если такого контейнера нет, объект помещается в новый (пустой) контейнер. Докажите, что количество контейнеров, выдаваемое этим эвристическим алгоритмом, превышает оптимальное не более чем в два раза.

8. *[5]* Для только что описанного эвристического алгоритма «первый подходящий» приведите пример экземпляра задачи, решение которого дает количество контейнеров, превышающее оптимальное максимум в 5/3 раз.

9. *[5]* Для заданного неориентированного графа $G = (V, E)$, каждый узел которого имеет степень $\leq d$, разработайте эффективный алгоритм вычисления независимого множества, размер которого, по крайней мере, в $1/(d + 1)$ раза меньше, чем размер наибольшего независимого множества.

10. *[5]* Целью задачи раскраски вершин графа $G = (V, E)$ является назначение цветов вершинам множества $V$ таким образом, чтобы вершины на концах каждого ребра были окрашены в разные цвета. Предоставьте алгоритм для раскраски графа $G$ не более чем $\Delta + 1$ разными красками, где $\Delta$ представляет максимальную степень вершин графа $G$.

11. *[5]* Продемонстрируйте, что можно решить любую головоломку судоку, вычислив минимальную вершинную раскраску конкретного, должным образом созданного графа, содержащего (9×9)+9 вершин.

## Комбинаторная оптимизация

Для каждой из приведенных далее задач разработайте и реализуйте эвристический алгоритм имитации отжига для получения приемлемых решений. Дайте оценку практической работе вашей программы.

12. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи минимизации ширины полос, рассматриваемой в *разд. 16.2.*

13. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи максимальной приемлемости, рассматриваемой в *разд. 17.10.*

14. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи поиска максимальной клики, рассматриваемой в *разд. 19.1.*

15. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи минимальной вершинной раскраски, рассматриваемой в *разд. 19.7.*

16. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи минимальной реберной раскраски, рассматриваемой в *разд. 19.8.*

17. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи поиска минимального разрывающего циклы множества вершин, рассматриваемой в *разд. 19.11.*

18. *[5]* Разработайте и реализуйте эвристический алгоритм для решения задачи покрытия множества, рассматриваемой в *разд. 21.1.*

## «Квантовые» вычисления

19. *[5]* Дана $n$-кубитовая «квантовая» система $Q$, у которой начальные вероятности всех ее $N = 2^n$ состояний одинаковые и составляют $p(i) = 1/2^n$. Предположим, операция `Jack(Q, 0^n)` удваивает вероятность состояния, в которой значения всех кубитов равны нулю. Сколько нужно выполнить таких операций `Jack`, чтобы вероятность выборки этого нулевого состояния составляла $\geq 1/2$?

20. *[5]* Для задачи выполнимости создайте: (a) экземпляр задачи из $n$ переменных, для которого есть ровно одно решение, и (b) экземпляр задачи из $n$ переменных, для которого есть ровно $2^n$ разных решений.

21. *[3]* Дана последовательность первых десяти чисел, кратных 11, т. е. 11, 22, 33, ..., 110. Из этой последовательности произвольно выбираем два числа ($x$ и $y$). Какова вероятность, что нод($x$, $y$) = 11?

22. *[8]* На веб-сайте квантовых вычислений компании IBM (**https://www.ibm.com/quantum-computing/**) предоставляется возможность программирования эмулятора квантовых вычислений. Изучите пример программы квантовых вычислений, запустите ее на исполнение и наблюдайте за результатами.

## LeetCode

1. **https://leetcode.com/problems/split-array-with-same-average/**
2. **https://leetcode.com/problems/smallest-sufficient-team/**
3. **https://leetcode.com/problems/longest-palindromic-substring/**

## HackerRank

1. **https://www.hackerrank.com/challenges/mancala6/**
2. **https://www.hackerrank.com/challenges/sams-puzzle/**
3. **https://www.hackerrank.com/challenges/walking-the-approximate longest-path/**

## Задачи по программированию

Эти задачи доступны на сайте **https://onlinejudge.org**:

1. «Euclid Problem», глава 7, задача 10104.
2. «Chainsaw Massacre», глава 7, задача 10043.
3. «Hotter Colder», глава 7, задача 10084.
4. «Useless Tile Packers», глава 7, задача 10065.

# Как разрабатывать алгоритмы?

Разработка правильного алгоритма для того или иного приложения в значительной степени является творческой деятельностью. Мы получаем задачу и придумываем решение для нее. Пространство выбора при разработке алгоритмов огромно, и у вас есть много возможностей допустить ошибку.

Цель этой книги состоит в том, чтобы повысить вашу квалификацию разработчика алгоритмов. Представленные в *первой части* этой книги методы содержат идеи, лежащие в основе всех комбинаторных алгоритмов. А каталог задач во *второй ее части* поможет вам моделировать свои приложения, предоставляя информацию о соответствующих задачах. Однако успешному разработчику алгоритмов требуется нечто большее, чем книжные знания. Он должен обрести определенный склад ума, позволяющий найти правильный подход к решению задач. Этому трудно научиться по книгам, но обладание таким качеством — необходимое условие.

Ключом к разработке алгоритмов (или любой другой деятельности, связанной с решением задач) является умение задавать самому себе вопросы, чтобы направлять свой процесс мышления: «Что будет, если я попробую такой способ? А если теперь попробовать другой?» Когда на каком-либо этапе вы столкнетесь с затруднениями, лучше всего перейти ко второму вопросу. Самым полезным участником любой мозговой атаки является тот, кто постоянно задает вопрос: «А почему нельзя применить такой-то способ?», а не тот, кто дает ответ на этот вопрос. В конце концов, первый участник предложит подход, который второй не сможет забраковать.

С этой целью мы предоставляем последовательность вопросов, которая должна направлять вашу деятельность в поиске правильного алгоритма для решения стоящей перед вами задачи. Но чтобы эти вопросы были эффективными, мало их задавать, нужно отвечать на них. Важно тщательно прорабатывать вопросы, записывая их в журнал. Правильным ответом на вопрос: «Можно ли сделать это таким способом?» является не просто: «Нет», а «Нет, потому что...». Четко излагая свои мысли по поводу отрицательного результата, вы сможете проверить, не упустили ли вариант, о котором просто не подумали. Поразительно, как часто причиной отсутствия у вас правильного объяснения является ошибочное умозаключение.

В процессе разработки важно осознавать разницу между *стратегией* и *тактикой* и постоянно помнить о ней. Стратегия — это поиск общей картины, основы, на которой строится решение. А тактика используется для решения второстепенных вопросов на пути к главной цели. В процессе решения задач важно постоянно проверять, на правильном ли уровне находятся ваши рассуждения. Если у вас нет глобального плана (т. е. стратегии) для решения поставленной перед вами задачи, то нет никакого смысла размышлять о правильной тактике. Примером стратегического вопроса явля-

ется: «Могу ли я смоделировать это приложение в виде задачи создания алгоритма на графах?» А тактическим вопросом может быть: «Какую структуру данных следует использовать для представления моего графа — список смежности или матрицу смежности?» Конечно же, тактические решения критичны для качества конечного решения, но их можно должным образом оценить только в свете успешной стратегии.

У слишком многих людей при необходимости решить задачу пропадают все мысли. Прочитав задачу, они приступают к ее решению и вдруг осознают, что *не знают, что делать дальше*. Чтобы не попадать в такие ситуации, следуйте списку вопросов, приведенных далее в этой главе и в большинстве разделов каталога задач. Они *подскажут* вам, что делать дальше.

Очевидно, что чем больше у вас имеется опыта по использованию методов разработки алгоритмов — таких как динамическое программирование, алгоритмы на графах и структуры данных, тем с большим успехом сможете вы следовать списку приведенных здесь вопросов. *Первая часть* книги и предназначена для развития и укрепления этих технических основ. Но, независимо от уровня вашей технической подготовки, всегда будет полезно следовать нашему списку. Самые первые и наиболее важные вопросы в этом списке помогут вам добиться глубокого понимания решаемой задачи и не потребуют никакого специального опыта.

Сам список вопросов создан под влиянием отрывка из замечательной книги о программе космических исследований США под названием «The Right Stuff» («Правильный материал») ([Wol79]). В этом отрывке описывались радиопереговоры пилотов-испытателей непосредственно перед крушением самолета. Можно было бы ожидать, что в такой ситуации они впадали в панику, но вместо этого пилоты выполняли действия из списка для нештатных ситуаций: «Проверил закрылки. Проверил двигатель. Крылья в порядке. Сбросил...» Эти пилоты обладали «Правильным материалом» и благодаря этому иногда им удавалось избежать гибели.

Я надеюсь, что моя книга и список вопросов помогут вам обрести «Правильный материал» разработчика алгоритмов. Я также рассчитываю, что это поможет вам избежать крушений в вашей деятельности.

## 13.1. Список вопросов для разработчика алгоритмов

1. Понимаю ли я задачу?
   - Из чего состоит вход?
   - Какой именно требуется результат?
   - Могу ли я создать достаточно небольшой входной экземпляр, чтобы его можно было решить вручную? Что будет, если я попытаюсь решить его?
   - Насколько важно для моего приложения всегда получать оптимальный результат? Устроит ли меня результат, близкий к наилучшему?
   - Каков размер типичного экземпляра решаемой задачи? 10 элементов? 1000 элементов? 1 000 000 элементов? Еще больше?

- Насколько важной является скорость работы приложения? Задачу нужно решить за одну секунду, одну минуту, один час или за один день?
- Сколько времени и усилий могу я вложить в реализацию? Буду ли я ограничен простыми алгоритмами, которые можно реализовать в коде за один день, или у меня будет возможность поэкспериментировать с несколькими подходами и посмотреть, какой из них лучше?
- Задачу какого типа я пытаюсь решить? Численную задачу, задачу теории графов, геометрическую задачу, задачу со строками или задачу на множестве? Какая формулировка выглядит самой легкой?

2. Могу ли я найти простой детерминированный или эвристический алгоритм для решения стоящей передо мной задачи?
   - Можно ли *правильно* решить задачу методом полного перебора всех возможных решений?
     - Если можно, то почему я уверен, что этот алгоритм всегда выдает правильное решение?
     - Как оценить качество созданного решения?
     - Каким будет время исполнения этого простого, но медленного алгоритма: полиномиальным или экспоненциальным? Настолько ли проста моя задача, чтобы ее можно было решить методом полного перебора?
     - Уверен ли я, что моя задача достаточно четко определена, чтобы правильное решение было *возможно*?
   - Можно ли решить эту задачу, постоянно применяя какое-либо простое правило, — например, выбирая в первую очередь наибольший элемент, наименьший элемент или произвольный элемент?
     - Если можно, то при каком входе этот эвристический алгоритм работает хорошо? Соответствует ли вход такого типа данным, актуальным для моего приложения?
     - При каком входе этот эвристический алгоритм работает плохо? Если не удастся найти примеры плохих входных экземпляров, то могу ли я показать, что алгоритм всегда работает хорошо?
     - Сколько времени требуется моему эвристическому алгоритму, чтобы выдать ответ? Можно ли его реализовать простым способом?

3. Есть ли моя задача в каталоге задач этой книги?
   - Что известно о моей задаче? Существует ли реализация решения, которую я могу использовать?
   - Искал ли я свою задачу в правильном месте каталога? Просмотрел ли я все рисунки? Проверил ли я все возможные ключевые слова в алфавитном указателе?
   - Существуют ли в Интернете ресурсы, имеющие отношение к моей задаче? Искал ли я в Google и Google Scholar? Посетил ли я веб-сайт этой книги — **www.algorist.com**?

4. Существуют ли специальные случаи стоящей передо мной задачи, которые я знаю, как решить?
   - Можно ли эффективно решить задачу, игнорируя некоторые входные параметры?
   - Станет ли задача проще, если присвоить некоторым входным параметрам тривиальные значения, — такие как 0 или 1?
   - Сумею ли я упростить задачу до такой степени, что *смогу* решить ее эффективно? Почему алгоритм для решения этого специального случая нельзя обобщить на более широкий диапазон входных экземпляров?
   - Не является ли моя задача специальным случаем одной из общих задач каталога?
5. Какая из стандартных парадигм разработки алгоритмов наиболее соответствует моей задаче?
   - Существует ли набор элементов, который можно отсортировать по размеру или какому-либо ключу? Способствует ли отсортированный набор данных более легкому решению задачи?
   - Можно ли разбить задачу на две меньших задачи — например, посредством двоичного поиска? Поможет ли метод разделения элементов на большие и маленькие или правые и левые? Может быть, стоит попробовать алгоритм типа «разделяй и властвуй»?
   - Не обладают ли объекты входного множества естественным порядком слева направо — как, например, символы в строке, элементы перестановки или листья дерева с корнем? Могу ли я применить динамическое программирование, чтобы воспользоваться этим упорядочиванием?
   - Нет ли повторяющихся операций, таких как поиск наименьшего/наибольшего элемента? Могу ли я применить специальные структуры данных для ускорения этих операций — например, словари/таблицы хеширования или пирамиды/очереди с приоритетами?
   - Могу ли я применить случайную выборку следующего объекта? Могу ли я создать несколько произвольных конфигураций и выбрать наилучшую из них? Могу ли я применить какой-либо эвристический метод поиска наподобие метода имитации отжига, чтобы выделить хорошее решение?
   - Могу ли я сформулировать свою задачу как линейную программу или как целочисленную программу?
   - Имеет ли моя задача какое-либо сходство с задачей выполнимости, задачей коммивояжера или какой-либо другой NP-полной задачей? Может ли задача быть NP-полной и вследствие этого не иметь эффективного алгоритма решения? Есть ли эта задача в списке из приложения к книге [GJ79]?
6. Я все еще не знаю, что делать?
   - Могу и хочу ли я потратить деньги, чтобы нанять специалиста (например, автора этой книги), чтобы он пояснил мне, что делать? Если можете, то ознакомьтесь со списком профессиональных консалтинговых услуг в *разд. 22.4*.

- Снова пройдитесь по этому списку вопросов с самого начала. Изменились ли ответы на некоторые вопросы при новом проходе по списку?

Решение задач не является точной наукой. Отчасти это искусство, а отчасти — ремесло. Навык решения задач — это один из полезнейших навыков, и его стоит приобрести. Моей любимой книгой по решению задач является книга «How to Solve It» («Как решить задачу») [Pol57], содержащая каталог методов решения задач. Я обожаю его просматривать — как с целью решения конкретной задачи, так и просто ради удовольствия.

## 13.2. Подготовка к собеседованию в технологических компаниях

Я надеюсь, что вы получаете удовольствие от чтения этой книги и начинаете понимать достоинства и мощь алгоритмического мышления. Но я подозреваю, что многих из моих читателей побудил к этому больше страх перед собеседованиями при приеме на работу, чем врожденная любовь к алгоритмам. Поэтому здесь я привожу несколько кратких советов, которые помогут вам в таких собеседованиях при приеме на работу в технологических компаниях.

Прежде всего следует понимать, что по теме разработки алгоритмов люди знают то, чему они учились в течение довольно длительного времени, поэтому зубрежка ночь напролет перед собеседованием вряд ли во многом вам поможет. Я полагаю, что узнать что-либо ценное о разработке алгоритмов можно, только посвятив серьезному изучению материала этой книги, по крайней мере, одну неделю. Излагаемый в ней материал самодостаточно полезен и стоит изучения ради его самого, а не для зазубривания к экзамену с тем, чтобы забыть его на следующий же день. При правильном подходе к изучению материала книги методы разработки алгоритмов обычно хорошо закрепляются в памяти, поэтому имеет смысл уделить работе с этой книгой достаточное время.

Задачи по разработке алгоритмов имеют склонность просачиваться в процесс собеседования двумя путями: при предварительной проверке знаний по написанию кода и при решении соискателем задач на классной доске. Крупные технологические компании привлекают такое большое количество соискателей, что первый раунд проверки часто чисто механический: можете ли вы решить какую-либо задачу по программированию на том или ином интернет-сервисе по набору персонала типа HackerRank (**https://www.hackerrank.com/**). По этим задачам программирования проверяются скорость и правильность кодирования для отсеивания наименее обещающих кандидатов.

Но уровень успеха решения этих задач программирования повышается при постоянной практике. Для этого студенты могут попытаться попасть в команду ACM ICPC (Association for Computing Machinery International Collegiate Programming Contest — Международная студенческая олимпиада по программированию Ассоциации по вычислительной технике) своего учебного заведения. Каждая такая команда состоит из трех членов, которые совместно должны решить от пяти до десяти задач по программированию в течение пяти часов. Эти задачи часто алгоритмического типа и обычно интересные. Членство в такой команде принесет много пользы, даже если ваша команда и не попадет на региональную олимпиаду.

Для самообучения я могу порекомендовать проверить себя в решении задач кодирования на сайтах для подготовки к техническим собеседованиям с автоматической оценкой ответов — например, того же HackerRank (**https://www.hackerrank.com**) или LeetCode (**https://leetcode.com**). Кстати, в конце каждой главы я рекомендую соответствующие задачи по программированию на каждом из этих двух сайтов. Начинайте с простых задач, постепенно усложняя их, и занимайтесь их решением, чтобы получать от этого удовольствие. Но сначала определите свое слабое звено: правильность определения граничных условий или ошибки в самом алгоритме, а затем практикуйтесь, чтобы улучшить свои навыки в этой области. Я могу скромно порекомендовать свою книгу «Programming Challengers»[1] [SR03], которая предназначена для использования в качестве учебного пособия при решении подобных задач по программированию. Если вам нравится книга, которую вы читаете сейчас, то вполне возможно, что и другая моя книга окажется для вас полезной.

Соискатели, прошедшие предварительный отбор, приглашаются на интервью: или дистанционно по видеоканалу, или с личным присутствием. Здесь вас вполне могут попросить решить несколько алгоритмических задач по выбору интервьюера на классной доске. Обычно это задачи, подобные упражнениям, приводящимся в конце каждой главы этой книги. Некоторые из этих упражнений так и называются: «*Задачи, предлагаемые на собеседовании*», т. к., по слухам, именно они входят в арсенал некоторых технологических компаний. Тем не менее, как для самообучения, так и для подготовки к собеседованию для приема на работу, полезны все упражнения из этой книги.

Какими должны быть ваши действия у классной доски, чтобы выглядеть так, как будто вы знаете, о чем говорите? Прежде всего, я рекомендую задавать достаточное количество уточняющих вопросов, чтобы уверенно понять суть поставленной перед вами задачи. Скорее всего, вы не получите много, если вообще сколько-либо, очков за правильное решение неправильно понятой задачи. Также я настоятельно рекомендую сначала предоставить простой, медленный и правильный алгоритм, прежде чем пытаться блеснуть своей изобретательностью. Только после этого можно и нужно посмотреть, можете ли вы сделать что-то получше. Обычно интервьюеры хотят увидеть, как вы думаете, и их интересует не столько сам конечный алгоритм, сколько ваш активный процесс мышления.

Следует отметить одно интересное обстоятельство: мои студенты, которые проходили такие собеседования, часто сообщают, что их интервьюеры давали им неправильные решения! Работа в хорошей компании никого автоматически не превращает в эксперта по алгоритмам. Иногда интервьюеры просто задают вопросы, список которых им предоставили другие люди, поэтому пусть это вас не пугает. Просто старайтесь решить поставленные перед вами задачи наилучшим образом, на который вы способны, и этого, скорее всего, будет достаточно.

Наконец, я должен кое в чем признаться. В течение многих лет я работал в должности руководителя исследовательских работ в стартапной компании General Sentiment и проводил собеседования со всеми разработчиками, которые хотели работать у нас.

---

[1] Доступен русский перевод этой книги: Стивен С. Скиена и Мигель А. Ревилла «Олимпиадные задачи по программированию».

Большинство из них знали, что я автор этой книги, и боялись, что я устрою им допрос с пристрастием. Но я никогда не задал никому из них ни одного вопроса по алгоритмическим головоломкам. Нам нужны были разработчики, которым были бы под силу сложные распределенные системы, а не те, кто умел бы решать головоломки. Я задавал много вопросов о том, с какой самой большой программой они работали и что именно они делали с ней, — чтобы получить представление об уровне сложности, на котором они могли работать, и о том, что им нравилось делать. Я очень горжусь теми замечательными людьми, которых мы приняли на работу в General Sentiment, — многие из них затем пошли работать в еще более интересные компании.

Конечно же, я настоятельно рекомендую *другим* компаниям продолжать строить свои собеседования на вопросах по алгоритмам. Чем больше компаний делают это, тем больше экземпляров этой книги будет продано.

Желаю вам всем удачи в ваших поисках работы, и пусть то, что вы почерпнете из моей книги, поможет вам как в этом, так и в дальнейшем профессиональном росте и карьере. Следите за мной в Твиттере по **@StevenSkiena**.

# ЧАСТЬ II

# Каталог алгоритмических задач

# ГЛАВА 14

# Описание каталога

*Вторая часть* книги содержит каталог алгоритмических задач, которые часто возникают на практике. Здесь приводится общая информация по каждой задаче и даются советы, как действовать, если та или иная задача возникнет в вашем приложении.

Как использовать этот каталог? Если вы знаете название своей задачи, то с помощью алфавитного указателя или оглавления найдите ее описание. Прочитайте весь материал, относящийся к задаче, поскольку он содержит ссылки на связанные с ней задачи. Пролистайте каталог, изучая рисунки и названия задач. Возможно, что-то покажется подходящим для вашего случая. Смело пользуйтесь алфавитным указателем — каждая задача внесена в него несколько раз под разными ключевыми словами.

Каталог содержит обширную информацию разных типов, которая никогда раньше не собиралась в одном месте.

Каждая задача сопровождается рисунком, представляющим входной экземпляр задачи и результат его решения. Эти стилизованные примеры иллюстрируют задачи лучше, чем формальные описания. Так, пример, касающийся минимального остовного дерева, демонстрирует кластеризацию точек с помощью минимальных остовных деревьев. Я надеюсь, что после беглого просмотра рисунков вы сможете найти то, что вам нужно.

Если вы нашли подходящую для вашего приложения задачу, прочитайте *раздел «Обсуждение»*. В нем описаны ситуации, в которых может возникнуть эта задача, и особые случаи входных данных, а также рассказано, какой результат можно разумно ожидать и, что еще более важно, как его добиться. Для каждой задачи приводятся краткое описание несложного решения, которым можно ограничиться в простых случаях, и ссылки на более мощные алгоритмы, чтобы вы могли ими воспользоваться, если первое решение окажется недостаточно эффективным.

Кроме того, приведены существующие программные реализации. Многие из этих процедур весьма удачны, и их код можно вставлять непосредственно в ваше приложение. Другие могут не отвечать требованиям промышленного применения, но я надеюсь, что они послужат хорошей моделью для вашей собственной реализации. Вообще, программные реализации приведены в порядке убывания их ценности, но при наличии бесспорно лучшего варианта я явно отмечаю его. В *главе 22* для многих из этих реализаций дается более подробная информация. Практически для всех реализаций на сайте этой книги (**www.algorist.com**) даются ссылки для их загрузки.

Наконец, в примечаниях к задаче рассказывается ее история и приводятся результаты, представляющие в основном теоретический интерес. Я постарался описать наилучшие известные результаты для каждой задачи и дать ссылки на работы по теоретическому и эмпирическому сравнению алгоритмов, если таковые имеются. Эта информация долж-

на представлять интерес для студентов, исследователей и практиков, не удовлетворившихся рекомендуемыми решениями.

## 14.1. Предостережения

Приведенный в этой части материал является каталогом алгоритмических задач, а не сборником рецептов, поскольку существует слишком много различных задач и вариантов их решений. Моя цель — указать вам правильное направление, чтобы вы могли решать свои задачи самостоятельно. Я попытался лишь очертить круг вопросов, с которыми вы столкнетесь в процессе решения задач.

- Для каждой задачи я дал рекомендацию, какой алгоритм следует использовать. Эти советы основаны на моем опыте и касаются приложений, которые я считаю типичными. При работе над книгой я полагал, что гораздо лучше дать конкретные рекомендации для типичных случаев, чем пытаться охватить все возможные ситуации. Если вы не согласны с моим советом, вы не обязаны следовать ему. Однако постарайтесь понять идеи, лежащие в основе моих советов, чтобы вы могли сформулировать причину, по которой ваши требования не соответствуют моим предположениям.
- Рекомендуемые мною реализации не обязательно являются исчерпывающими решениями вашей задачи. Некоторые программы пригодны только в качестве моделей для создания вашего собственного кода. Другие встроены в большие системы, и их может быть слишком трудно извлечь и исполнять отдельно. Все эти решения содержат ошибки. Многие из таких ошибок могут оказаться довольно серьезными, так что будьте начеку.
- Вы обязаны соблюдать условия лицензии любой реализации, которую вы используете в коммерческих целях. Многие из этих программ не являются свободно распространяемыми, и большинство имеет лицензионные ограничения. Подробности см. в *разд. 22.1*.
- Мне бы хотелось узнать о результатах, полученных вами при следовании моим рекомендациям, — как о положительных, так и об отрицательных. Особый интерес для меня представляет информация о других реализациях, неизвестных мне, но известных вам.

# Структуры данных

Структуры данных — это базовые конструкции для построения приложений. Чтобы максимально полно использовать возможности стандартных структур данных, необходимо иметь о них ясное представление. Поэтому мы сначала подробно рассмотрим различные структуры данных, а затем перейдем к обсуждению других задач.

На мой взгляд, наиболее важным аспектом этого раздела каталога станут предоставленные в нем ссылки на разные программные реализации и библиотеки структур данных. Хорошая реализация многих структур данных — дело далеко не тривиальное, поэтому программы, на которые даются ссылки, окажутся вам полезными в качестве моделей, даже если они и не в точности подходят для ваших задач. Некоторые важные структуры данных, такие как kd-деревья и суффиксные деревья, известны не настолько хорошо, насколько они того заслуживают. Будем надеяться, что этот раздел каталога добавит им известности.

Существует большое количество книг по элементарным структурам данных. На мой взгляд, лучшими из них являются следующие:

- [SW11] — подробное введение в алгоритмы и структуры данных отличающееся удачными рисунками, показывающими алгоритмы в действии. Имеются версии книги для языков C, C++ и Java;
- [Wei11] — хороший учебник, с упором на структуры данных в большей степени, чем на алгоритмы. Имеются версии книги для языков Java, C, C++ и Ada;
- [GTG14] — версия книги для Java с активным использованием авторской библиотеки структур данных JDSL (Java Data Structures Library);
- [Bra08] — хорошее рассмотрение более продвинутых структур данных, чем те, которые рассматриваются в других учебниках с реализацией на языке C++.

Книга [MS18] содержит всесторонний обзор современных исследований в области структур данных. Читатель, знакомый лишь с основными понятиями из этой области, будет удивлен объемом проводимых исследований.

## 15.1. Словари

**Вход.** Множество из $n$ записей, каждая из которых идентифицируется одним или несколькими ключами.

**Задача.** Создать и поддерживать структуру данных для эффективного поиска, вставки и удаления записи, связанной с ключом запроса $q$ (рис. 15.1).

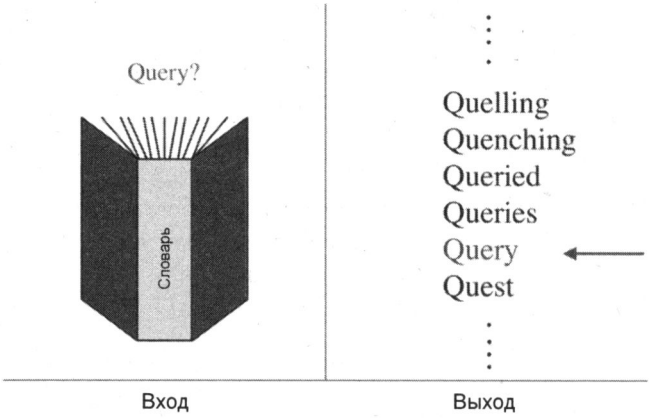



**Рис. 15.1.** Поиск в словаре

### Обсуждение

Абстрактный тип данных «словарь» (dictionary) является одной из наиболее важных структур в теории вычислительных систем. Для реализации словарей было предложено несколько десятков структур данных, включая хеш-таблицы, списки с пропусками и двоичные деревья поиска. Это означает, что выбрать наилучшую структуру данных не всегда легко. *Однако на практике важнее избежать применения неподходящей структуры данных, чем найти самую лучшую.*

Вы должны тщательно изолировать словарную структуру данных от ее интерфейса. Используйте явные вызовы методов или процедур для инициализации, поиска и модифицирования структуры данных, а не внедряйте их в код структуры. Такой подход не только позволяет получить более аккуратную программу, но также облегчает проведение экспериментов с разными реализациями для оценки их производительности. Не озабочивайтесь по поводу неизбежных накладных расходов по вызову процедур. Если время работы вашего приложения является очень серьезным фактором, то такие эксперименты важны тем, что помогут вам выбрать правильную реализацию словаря, чтобы минимизировать его.

Выбирая структуру данных для словаря, ответьте на несколько вопросов.

- *Сколько элементов будет содержать структура данных?*

  Известно ли их количество заранее? Достаточно ли будет для решения вашей задачи простой структуры данных, или же размер задачи настолько велик, что следует беспокоиться о нехватке памяти или производительности виртуальной памяти?

- *Известна ли относительная частота операций вставки, удаления и поиска?*

  *Статические* структуры данных (например, упорядоченные массивы) достаточны для приложений, в которых структура данных не подвергается изменениям после ее первоначального создания. *Полудинамические структуры данных*, поддерживающие только операцию вставки и не допускающие удаление данных, реализовать значительно проще, чем полностью динамические.

- *Будет ли обращение к ключам единообразным и случайным?*

  Во многих приложениях поисковые запросы имеют асимметричное распределение обращений, т. е. некоторые элементы пользуются большей популярностью, чем другие. Кроме того, запросы часто обладают свойством временнóй локальности. Иными словами, запросы периодически приходят группами (кластерами), а не через регулярные интервалы времени. В структурах данных типа косых деревьев можно воспользоваться асимметричным и кластеризированным пространством.

- *Критична ли скорость каждой отдельной операции или требуется минимизировать только общий объем работы, выполняемый всей программой?*

  Когда время реакции критично — например, в программе управления аппаратом искусственного кровообращения, время ожидания выполнения следующего шага не может быть слишком большим. А при выполнении множества запросов к базе данных — например, для выяснения, кто из нарушителей закона является политическим деятелем, скорость поиска конкретного законодателя не является критичной, если удается выявить их всех с минимальными общими затратами времени.

Представители современного поколения «объектно-ориентированных» программистов способны написать контейнерный класс не в большей степени, чем отремонтировать двигатель в своей машине. В этом нет ничего плохого — для большинства приложений вполне достаточно стандартных контейнеров. Тем не менее иногда полезно знать, что именно находится «под капотом»:

- *неупорядоченные списки или массивы.*

  Для небольших наборов данных самой простой в обслуживании структурой будет неупорядоченный массив. По сравнению с аккуратными и компактными массивами производительность связных структур может быть крайне неудовлетворительной. Но когда размер словаря превысит 50–100 элементов, линейное время поиска в списке или массиве сведет на нет все его преимущества.

  Особенно интересным и полезным вариантом является *самоорганизующийся список* (self-organizing list), в котором вставляемый или просматриваемый элемент перемещается в начало списка. Таким образом, если в ближайшем будущем к тому или иному ключу осуществляется повторное обращение, то этот ключ будет находиться вблизи начала списка и его поиск займет очень мало времени. Большинство приложений демонстрируют как неоднородную частоту обращений, так и пространственную локальность, поэтому среднее время успешного поиска в самоорганизующемся списке обычно намного меньше, чем в упорядоченном или неупорядоченном списке. Самоорганизующиеся структуры данных можно создавать из массивов точно так же, как из связных списков и деревьев;

- *упорядоченные списки или массивы.*

  Обслуживание упорядоченного списка обычно не стоит затрачиваемых усилий, поскольку эта структура данных не поддерживает двоичный поиск, — если только вы не пытаетесь избежать создания дубликатов. Использование упорядоченного массива будет уместным только в тех случаях, когда не требуется выполнять большое количество вставок и/или удалений;

♦ *хеш-таблицы.*

Для приложений, работающих с количеством элементов в диапазоне от умеренного до большого, использование хеш-таблицы будет, скорее всего, правильным выбором. Хеш-функция устанавливает соответствие ключей (будь то строки, числа или что угодно другое) и целых чисел в диапазоне от 0 до $m - 1$. Создание хеш-таблицы сопровождается массивом из $m$ корзин, реализованных в виде неупорядоченного связного списка. Хеш-функция немедленно определяет корзину, содержащую искомый ключ. Если используемая хеш-функция распределяет ключи достаточно равномерно в хеш-таблице достаточно большого размера, то каждая корзина будет содержать очень небольшое количество элементов, что делает линейные поиски приемлемыми. Вставка и удаление элемента из хеш-таблицы сводится к вставке и удалению из корзины/списка. Хеширование подробно обсуждалось в *разд. 3.7.*

В большинстве приложений производительность хорошо отлаженной хеш-таблицы будет превосходить производительность упорядоченного массива. Но прежде чем приступить к реализации хеш-таблицы, ответьте на несколько вопросов.

- *Каким образом обрабатывать коллизии?*

  Использование открытой адресации вместо корзин позволит получить небольшие таблицы с хорошей производительностью, но при слишком высоком коэффициенте нагрузки (отношение заполненности к емкости) производительность хеш-таблицы будет понижаться.

- *Каким должен быть размер таблицы?*

  Если задействуются корзины, то значение $m$ должно быть приблизительно равным максимальному количеству элементов, которое вы планируете поместить в таблицу. Если применяется открытая адресация, то значение $m$ должно превышать ожидаемое максимальное количество элементов, по крайней мере, на 30–50%. Выбор в качестве $m$ простого числа компенсирует возможные недостатки неудачной хеш-функции.

- *Какую хеш-функцию использовать?*

  Для строк должна работать, например, такая формула:

$$H(S) = \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i) (\mathrm{mod}\, m),$$

  где $\alpha$ — размер алфавита; $char(x)$ — функция, которая для каждого символа $x$ возвращает его код. Для реализации эффективного вычисления этой хеш-функции используйте правило Горнера (или заранее вычисляйте значения $\alpha^x$), как показано в *разд. 16.9*. Версия этой хеш-функции (см. *разд. 6.7*) обладает одним интересным свойством, заключающимся в том, что хеш-коды последовательных окон строки размером в $k$ символов можно вычислять за постоянное время, а не за время $O(k)$.

При оценке эффективности используемой хеш-функции изучите статистические данные относительно распределения ключей по корзинам, чтобы выяснить, насколько она *действительно* является однородной. Вероятно, что самая первая вы-

бранная хеш-функция окажется не самой лучшей. Использование неудачной хеш-функции может заметно снизить производительность любого приложения;

- *двоичные деревья поиска.*

  Двоичные деревья представляют собой элегантные структуры данных, которые поддерживают быстрое исполнение операций вставки, удаления и запросов (см. *разд. 3.4*). Главное отличие между разными типами деревьев проявляется в том, выполняется ли их балансировка явным образом после операций вставки и удаления и в способе ее осуществления. В простых *произвольных деревьях поиска* каждый узел вставляется в некий лист дерева и повторная балансировка не осуществляется. Хотя такие деревья имеют хорошую производительность в случае произвольных вставок, большинство приложений не обеспечивают случайность вставок. И действительно, несбалансированные деревья поиска, созданные вставками элементов в отсортированном порядке, никуда не годятся, поскольку их производительность ухудшается до производительности связных списков.

  Структура сбалансированных деревьев поиска обновляется с помощью локальных операций *ротации*, которые перемещают удаленные узлы ближе к корню, таким образом поддерживая упорядоченную структуру дерева. В настоящее время такие сбалансированные деревья поиска, как AVL и 2/3, считаются устаревшими, и наибольшее предпочтение отдается *красно-черным деревьям*. Особенно интересной самоорганизующейся структурой данных являются *косые деревья*. В них ключи, к которым было осуществлено обращение, перемещаются в корень посредством операции ротации. В результате часто используемые или недавно посещенные узлы находятся в верхней части дерева, что позволяет быстрее производить поиск.

  Итак, какой тип дерева лучше всего подойдет для вашего приложения? Наверное, тот, для которого у вас имеется наилучшая реализация. Не так важен тип используемого сбалансированного дерева, как профессионализм программиста, реализовавшего его;

- *B-деревья.*

  Для наборов данных настолько больших, что они не помещаются в оперативную память, наилучшим выбором будет какой-либо тип B-дерева. Когда структура данных хранится вне оперативной памяти, время поиска увеличивается на несколько порядков. Подобное падение производительности в меньшем масштабе может наблюдаться в системах с современной архитектурой кэша, поскольку кэш работает намного быстрее, чем оперативная память.

  Идея, лежащая в основе B-дерева, состоит в сворачивании нескольких уровней двоичного дерева поиска в один большой узел, чтобы можно было выполнить операцию, эквивалентную нескольким шагам поиска, прежде чем потребуется новое обращение к диску. С помощью B-деревьев можно получать доступ к огромному количеству ключей, выполняя небольшое количество обращений к диску. Чтобы максимально эффективно использовать B-деревья, нужно понимать, как взаимодействуют внешние запоминающие устройства с виртуальной памятью, — в частности, такие аппаратные характеристики, как размер страницы и виртуальное/реальное адресное пространство. *Нечувствительные к кэшированию алгоритмы* (cache-oblivious algorithms) позволяют уменьшить значение этих факторов.

Даже в случае наборов данных небольшого размера результатом использования файлов подкачки может быть неожиданно низкая производительность, поэтому необходимо следить за интенсивностью обращений к жесткому диску, чтобы получить дополнительную информацию для принятия решения, стоит ли использовать B-деревья;

- *списки с пропусками.*

  Списки с пропусками представляют собой иерархическую структуру упорядоченных связных списков, в которых решение, копировать ли элемент в список более высокого уровня, принимается случайным образом. В структуре задействовано примерно $\lg n$ списков, и размер каждого из них приблизительно вдвое меньше размера списка, расположенного уровнем выше. Поиск начинается в самом коротком списке. Искомый ключ находится в интервале между двумя элементами и потом ищется в списке большего размера. Каждый интервал поиска содержит ожидаемое постоянное количество элементов в каждом списке, при этом общее ожидаемое время исполнения запроса составляет $O(\lg n)$. Основные достоинства списков с пропусками по сравнению со сбалансированными деревьями — легкость анализа и реализации.

### Реализации

Современные языки программирования содержат библиотеки подробных и эффективных реализаций контейнеров. В настоящее время с большинством компиляторов для языка C++ поставляется библиотека STL (Standard Template Library). Более подробное руководство по использованию библиотеки STL и стандартной библиотеки C++ можно найти в книгах [Jos12], [Mey01] и [MDS01]. Небольшая библиотека структур данных Java Collections (JC) входит в стандартный пакет утилит Java java.util.

Библиотека LEDA (см. *раз. 22.1.1*) предоставляет полную коллекцию словарных структур данных, реализованных на языке C++, включая хеш-таблицы, совершенные хеш-таблицы, B-деревья, красно-черные деревья, деревья случайного поиска и списки с пропусками. В результате экспериментов, описанных в книге [MN99], было определено, что наилучшим вариантом для словарей являются хеш-таблицы, а списки с пропусками и (2,4)-деревья (частный случай B-деревьев) являются наиболее эффективными древоподобными структурами.

#### Примечания

В книге [Knu97a] представлено наиболее подробное описание и анализ основных словарных структур данных. Но в ней отсутствуют некоторые современные структуры данных, такие как красно-черные и косые деревья. Чтение книг Дональда Кнута послужит хорошим введением в предмет для всех изучающих теорию вычислительных систем.

В книге [MS18]] дан всесторонний обзор современных исследований в области структур данных. Другие исследования представлены в книгах [MT90b] и [GBY91]. Хорошими учебниками по словарным структурам данных являются [Sed98], [Wei11] и [GTG14]. Мы отсылаем читателя к этим работам, чтобы не приводить здесь ссылки на описания структур данных, обсуждавшихся ранее.

Соревнование DIMACS в 1996 году было посвящено реализациям элементарных структур данных, включая словари (см. [GJM02]). Наборы данных и код можно загрузить с вебсайта **http://dimacs.rutgers.edu/Challenges**.

Для многих задач стоимость обмена данными между различными запоминающими устройствами (оперативной памятью, кэшем или диском) превышает стоимость собственно вычислений. В каждой операции по пересылке данных перемещается один блок размером $b$, поэтому эффективные алгоритмы стремятся минимизировать количество перемещений блоков. Исследование сложности фундаментальных алгоритмов и структур данных для этой модели внешней памяти представлено в журнале [Vit01]. *Нечувствительные к кэшированию* структуры данных дают гарантию производительности для такой модели, не требуя при этом явной информации о параметре размера блока $b$. Таким образом, хорошую производительность можно получить на любой машине, не прибегая к специфическим для ее архитектуры настройкам. Превосходное исследование структур данных, не чувствительных к кэшированию, приведено в [ABF05] и [Dem02].

Косые деревья и другие современные структуры данных изучались с применением *амортизированного анализа*, при котором устанавливается верхняя граница для общего времени исполнения любой последовательности операций. При амортизированном анализе одна операция может быть очень дорогой, но только потому, что ее стоимость компенсируется низкими затратами на другие операции. Структура данных, имеющая амортизированную сложность $O(f(n))$, является менее предпочтительной, чем структура с такой же сложностью в наихудшем случае (поскольку существует вероятность выполнения очень дорогой операции), но она все же лучше структуры с такой сложностью в среднем случае (т. к. амортизированная граница достигнет этого значения при любом входе).

### Родственные задачи

Сортировка (см. *разд. 17.1*), поиск (см. *разд. 17.2*).

## 15.2. Очереди с приоритетами

**Вход.** Набор записей, ключи которых полностью упорядочены.

**Задача.** Создать и поддерживать структуру данных для обеспечения быстрого доступа к *наименьшему* или *наибольшему* ключу (рис. 15.2).

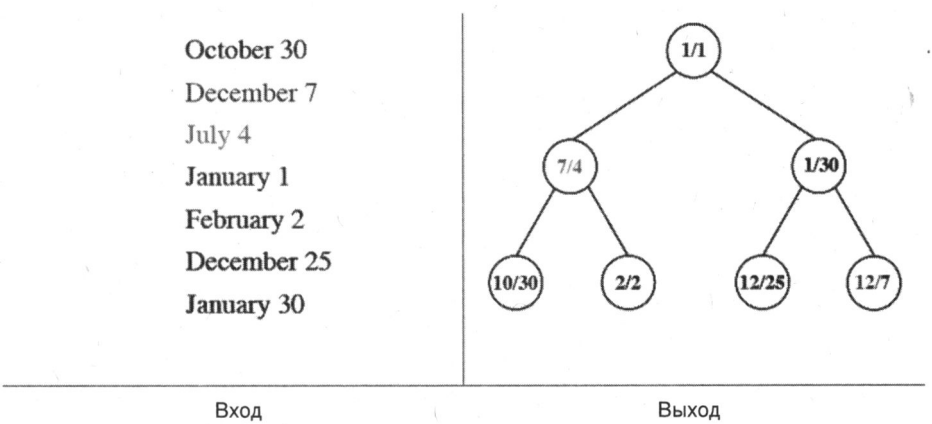



**Рис. 15.2.** Очередь с приоритетами

### Обсуждение

Очереди с приоритетами (priority queues) являются полезными структурами данных при моделировании поведения систем, особенно при сопровождении набора запланированных событий, упорядоченных по времени. Они получили такое название потому, что элементы из них можно извлекать не в зависимости от времени вставки (как в стеке или обычной очереди) и не в результате совпадения ключа (как в словаре), а по их приоритету на извлечение.

Если после первоначального запроса приложение не будет вставлять новые элементы, то в явно выраженной очереди с приоритетами нет необходимости. Просто отсортируйте записи по приоритетам и выполняйте обработку сверху вниз, поддерживая при этом указатель на последнюю извлеченную запись. Такая ситуация возникает в алгоритме Крускала при построении минимального остовного дерева или при выполнении полностью распланированного сценария развития событий.

Но если операции вставки и удаления чередуются с запросами, понадобится настоящая очередь с приоритетами. Чтобы решить, какой тип очереди следует применить, ответьте на несколько вопросов.

- *Какие еще операции вам потребуются?*

  Будет ли выполняться поиск произвольных ключей или только наименьшего ключа? Будут ли удаляться произвольные элементы данных или просто будет повторно удаляться верхний или нижний элемент?

- *Известен ли заранее максимальный размер структуры данных?*

  Здесь вопрос заключается в том, можно ли будет выделить место для структуры данных заранее.

- *Можно ли повышать или понижать приоритет элементов, уже поставленных в очередь?*

  Возможность изменять приоритет элементов требует, кроме обнаружения наибольшего элемента, возможности извлекать элементы из очереди по ключу.

Приведем базовые реализации очереди с приоритетами:

- *упорядоченный массив или список.*

  Упорядоченный массив очень эффективен как для идентификации наименьшего элемента, так и для его удаления путем уменьшения значения максимального индекса. Но поддержание упорядоченности элементов замедляет вставку новых элементов. Упорядоченные массивы подходят для реализации очереди с приоритетами только тогда, когда в нее не будет осуществляться никаких вставок. Базовые реализации очередей с приоритетами рассматриваются в *разд. 3.5*.

- *двоичные пирамиды.*

  Эта простая и элегантная структура данных поддерживает как операцию вставки, так и операцию извлечения наименьшего элемента за время $O(\lg n)$. Пирамиды позволяют поддерживать явную структуру двоичного дерева в массиве, чтобы значение ключа корня любого поддерева было меньшим, чем любого из его потомков. Таким образом, наименьший ключ всегда находится вверху пирамиды. Новые эле-

менты можно добавлять, вставляя элемент в свободный лист дерева и перемещая его вверх до тех пор, пока он не займет правильное место при условии частичной упорядоченности. Реализация двоичной пирамиды на языке C рассматривается в *разд. 4.3.2*, а операция извлечения из нее наименьшего элемента — в *разд. 4.3.3*.

Двоичные пирамиды являются удачным выбором в том случае, когда известна верхняя граница количества элементов в очереди с приоритетами, поскольку размер массива нужно задавать при его создании. Но даже это ограничение можно обойти с помощью динамических массивов (см. *разд. 3.1.1*);

- *очередь с приоритетами, имеющая ограничение по высоте.*

  Эта структура данных на основе массива позволяет выполнять операции вставки и поиска наименьшего элемента за постоянное время при ограниченном диапазоне возможных значений ключей. Допустим, мы знаем, что все значения ключей будут целыми числами в диапазоне от 1 до $n$. Тогда можно создать массив из $n$ связных списков, где $i$-й список играет роль корзины, содержащей все элементы с ключом $i$. На наименьший непустой список будем поддерживать указатель *top*. Чтобы вставить в очередь с приоритетами элемент с ключом $k$, добавляем его к $k$-й корзине и присваиваем этому указателю значение $top = \min(top, k)$. Чтобы извлечь наименьший элемент, определяем первый элемент в корзине *top*, удаляем его и, если корзина стала пустой, перемещаем указатель *top* вниз.

  Очереди с приоритетами, имеющие ограничение по высоте, очень полезны для хранения вершин графа, упорядоченных по их степеням, при том что такое упорядочивание является одной из основных операций в алгоритмах на графах. Тем не менее при этом они применяются не так широко, как того заслуживают. Обычно этот тип очереди подходит для случаев небольших, дискретных диапазонов ключей;

- *двоичные деревья поиска.*

  Из двоичных деревьев поиска получаются эффективные очереди с приоритетами, поскольку самый левый лист всегда содержит наименьший элемент, а самый правый — наибольший. Чтобы найти наименьший (или наибольший) элемент, просто выполняется проход по указателям влево (или соответственно вправо) вниз до тех пор, пока указатель не станет нулевым. Двоичные деревья поиска оказываются наиболее подходящими, когда также требуется осуществлять и другие словарные операции, или в случае неограниченного диапазона ключей, когда максимальный размер очереди с приоритетами неизвестен заранее;

- *Фибоначчиевы и парные (pairing) пирамиды.*

  Такие сложные очереди с приоритетами предназначены для ускорения выполнения операции *уменьшения ключа*, которая понижает приоритет поставленного в очередь элемента. Подобная ситуация возникает, например, в вычислениях кратчайшего пути — при обнаружении более короткого, чем найденный ранее, маршрута к вершине $v$.

  Если их реализовать и использовать должным образом, то этот тип очереди с приоритетами может улучшить производительность при больших объемах вычислений.

### Реализации

Современные языки программирования содержат библиотеки подробных и эффективных реализаций очередей с приоритетами. Класс `PriorityQueue` библиотеки Java Collections (JC) входит в стандартный пакет утилит Java java.util. Методы `push`, `top` и `pop` шаблона `priority_queue` библиотеки STL языка C++ воспроизводят операции с пирамидами `insert`, `findmax` и `deletemax`. Более подробное руководство по использованию библиотеки STL можно найти в книгах [Mey01] и [MDS01].

Библиотека LEDA (см. *разд. 22.1.1*) содержит полную коллекцию очередей с приоритетами на языке C++, включая фибоначчиевы пирамиды, парные пирамиды, деревья ван Эмде Боаса и очереди с приоритетами, имеющие ограничение по высоте. В результате экспериментов, описанных в книге [MN99], было установлено, что простые двоичные пирамиды являются достаточно конкурентоспособными в большинстве приложений, при этом в прямом сравнении парные пирамиды показывают лучшую производительность, чем фибоначчиевы пирамиды. В статье [San00] изложено описание экспериментов, демонстрирующих, что производительность разработанной автором последовательной пирамиды (sequence heap), основанной на $k$-м слиянии, приблизительно вдвое лучше, чем хорошо реализованной двоичной пирамиды.

#### *Примечания*

В книге [MS18] приводится всесторонний обзор современного состояния дел в том, что касается очередей с приоритетами. Результаты экспериментальных сравнений структур данных, реализующих очереди с приоритетами, представлены в таких работах, как [CGS99], [GBY91], [Jon86], [LL96] и [San00].

Двусторонние очереди с приоритетами (double-ended priority queue) позволяют расширить набор операций с пирамидами, включив в него одновременно операции `find-min` и `find-max`. Обзор четырех разных реализаций двусторонних очередей с приоритетами приводится в [Sah05].

Очереди с приоритетами, имеющие ограничение по высоте, являются подходящими структурами данных для использования на практике, но они не гарантируют высокой производительности в наихудшем случае при неограниченных диапазонах ключей. Однако очереди с приоритетами ван Эмде Боаса (см. [vEBKZ77]) позволяют выполнять операции вставки, удаления, поиска и определения наибольшего и наименьшего элементов за время $O(\lg \lg n)$, где каждый ключ имеет значение от 1 до $n$.

Фибоначчиевы пирамиды (см. [FT87, BLT12]) поддерживают выполнение операций вставки и уменьшения ключа за амортизированное постоянное время, а выполнение операций извлечения наименьшего элемента и удаления — за время $O(\lg n)$. Постоянное время исполнения операции уменьшения ключа позволяет получить более быструю реализацию алгоритмов поиска кратчайшего пути и паросочетаний во взвешенных двудольных графах и минимальных остовных деревьях. Фибоначчиевы пирамиды на практике трудно реализовать, и время исполнения их операций содержит большие постоянные коэффициенты. Парные пирамиды поддерживают такие же пределы, но при меньших накладных расходах. Эксперименты с парными и другими пирамидами описаны в [LST14] и [SV87].

Пирамиды поддерживают частичное упорядочивание, которое можно создать за линейное количество операций сравнения. Обычные алгоритмы слияния с линейным временем исполнения для создания пирамид описаны в [Flo64]. В наихудшем случае достаточно $1{,}625n$ сравнений (см. [GM86]), а вообще для создания пирамиды необходимо от $1{,}5n$ до $O(\lg n)$ сравнений (см. [CC92]).

### Родственные задачи

Словари (см. *разд. 15.1*), сортировка (см. *разд. 17.1*), поиск кратчайшего пути (см. *разд. 18.4*).

## 15.3. Суффиксные деревья и массивы

**Вход.** Строка $S$.

**Задача.** Создать структуру данных, позволяющую быстро определять местоположение произвольной строки запроса $q$ в строке $S$ (рис. 15.3).

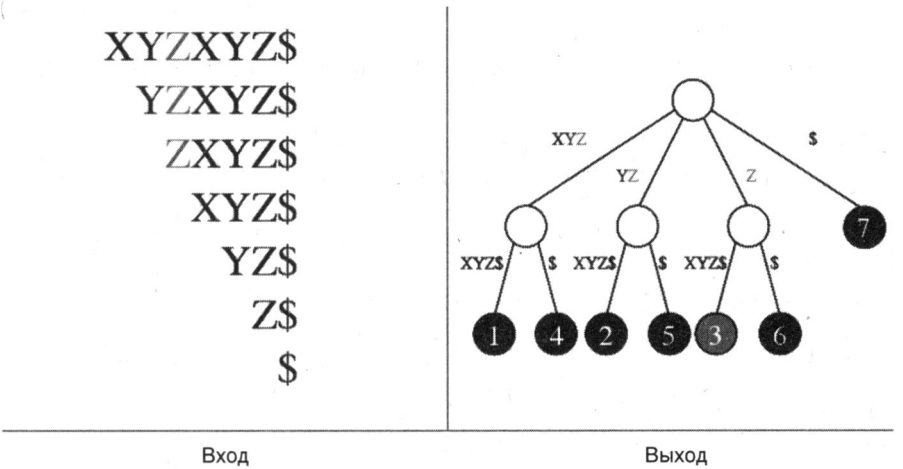



**Рис. 15.3.** Суффиксное дерево

### Обсуждение

Суффиксные деревья и массивы являются исключительно полезными структурами данных для элегантного и эффективного решения задач по обработке строк. Правильное применение суффиксных деревьев часто позволяет уменьшить время исполнения алгоритмов для обработки строк с $O(n^2)$ до линейного. Суффиксные деревья упоминались в истории из жизни, описанной в *разд. 3.9*.

Простейший экземпляр суффиксного дерева представляет собой простое *нагруженное дерево* (trie) из $n$ суффиксов строки $S$ длиной в $n$ символов. Нагруженным деревом называется древовидная структура, в которой каждое ребро представляет один символ, а корень — нулевую строку.

Каждый путь из корня определяет строку, описываемую символами, которые маркируют проходимые ребра. Каждый конечный набор слов определяет отдельное нагруженное дерево, а дерево для двух слов с общим префиксом разветвляется на первом несовпадающем символе. Каждый лист дерева обозначает конец строки. На рис. 15.4 показан пример простого нагруженного дерева.

Нагруженные деревья полезны для проверки на вхождение какой-либо строки запроса $q$ в набор строк. Обход нагруженного дерева выполняется с корня вдоль ветвей,

определяемых последующими символами строки $q$. Если какая-либо ветвь отсутствует в нагруженном дереве, то тогда строка $q$ не может быть элементом набора строк, определяемых этим нагруженным деревом. В противном случае строка *запроса* находится за $|q|$ сравнений символов, *независимо* от количества других строк, содержащихся в нагруженном дереве. Нагруженные деревья очень легко создавать (последовательно вставляя новые строки), и в них быстро выполняется поиск (простым проходом вниз по дереву), хотя они могут оказаться дорогими в смысле расхода памяти.

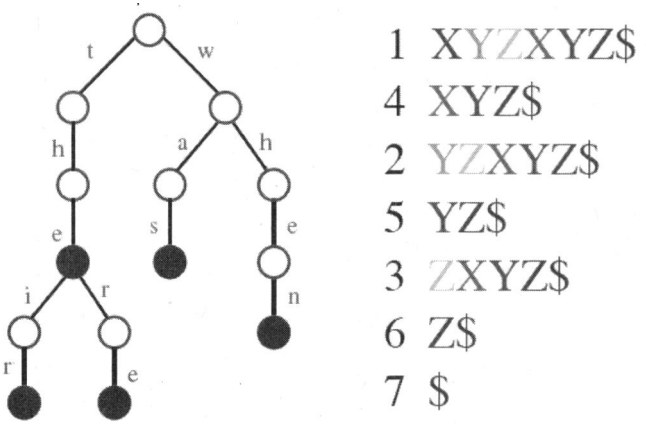



**Рис. 15.4.** Нагруженное дерево для строк `the`, `their`, `there`, `was` и `when` (*слева*) и соответствующий массив суффиксов `XYZXYZ$` (*справа*)

*Суффиксное дерево* — это просто нагруженное дерево всех суффиксов строки $S$. Суффиксное дерево позволяет проверить, является ли строка запроса $q$ подстрокой строки $S$, т. к. любая подстрока строки $S$ представляет собой префикс какого-либо суффикса. Время поиска — линейное по отношению к длине строки запроса $|q|$.

Но проблема здесь заключается в том, что создание полного суффиксного дерева таким способом может занять время $O(n^2)$ и, что еще хуже, такой же объем памяти, поскольку средняя длина $n$ суффиксов равна $n/2$. Впрочем, при рациональном подходе для представления полного суффиксного дерева затраты памяти будут линейными. Обратите внимание на то, что большинство узлов нагруженного суффиксного дерева находятся на простых неветвящихся путях между узлами дерева с полустепенью исхода $\geq 2$. Каждый из этих простых путей соответствует подстроке первоначальной строки. Сохранив первоначальную строку в массиве, любой такой свернутый путь можно представить начальным и конечным индексами массива, определяющего нашу подстроку. Таким образом, каждое ребро дерева маркируется с использованием только двух целых чисел, в результате чего всю информацию о полном суффиксном дереве можно сохранить в памяти объемом всего лишь $O(n)$. Такое свернутое суффиксное дерево показано на рис. 15.3 (*справа*).

Для создания свернутого дерева существуют алгоритмы, использующие дополнительные указатели, ускоряющие процесс построения. Время исполнения этих алгоритмов равно $O(n)$. Эти дополнительные указатели можно также задействовать для ускорения многих приложений, основанных на суффиксных деревьях.

Но для чего нужны суффиксные деревья? Далее приводится краткое описание вариантов применения суффиксных деревьев.

- *Поиск всех вхождений подстроки q в строке S.*

  Так же как и в случае с нагруженным деревом, мы можем выполнять обход, начиная с корня дерева по направлению к узлу $n_q$, связанному с подстрокой $q$. Позиции всех вхождений подстроки $q$ в строку $S$ представлены потомками узла $n_q$, которых можно определить посредством поиска в глубину, начиная с узла $n_q$. Используя свернутое суффиксное дерево, найти $k$ вхождений подстроки $q$ в строку $S$ можно за время $O(|q| + k)$.

- *Поиск самой длинной подстроки, общей для набора строк.*

  Создается одно свернутое суффиксное дерево, содержащее все суффиксы всех строк, в котором каждый лист дерева помечен его исходной строкой. В процессе обхода в глубину этого дерева мы помечаем каждый его узел информацией о длине его префикса и количестве разных строк, являющихся его потомками. По этой информации наилучший узел можно найти за линейное время.

- *Поиск самого длинного палиндрома в строке S.*

  *Палиндром* — это строка, которая читается одинаково в обоих направлениях, — например: *мадам*. Чтобы найти самый длинный палиндром в строке $S$, создаем суффиксное дерево, содержащее все суффиксы строки $S$ и строки, обратной $S$. Каждый лист этого дерева будет идентифицироваться его начальной позицией. Палиндром определяется любым узлом этого дерева, у которого из одной позиции выходят потомки, одинаковые в обоих направлениях.

Так как алгоритмы с линейным временем исполнения для создания суффиксных деревьев весьма нетривиальны, вместо самостоятельной разработки таких алгоритмов рекомендуется использовать существующие реализации. Но в качестве альтернативы можно задействовать суффиксные массивы, которые позволяют делать практически все то, что и суффиксные деревья, но при этом их легче реализовать.

В принципе, *суффиксный массив* представляет собой просто массив, содержащий все $n$ суффиксов строки $S$ в отсортированном порядке. Соответственно двоичного поиска строки $q$ в этом массиве оказывается достаточно для локализации префикса суффикса, совпадающего с подстрокой $q$, что гарантирует эффективный поиск подстроки за $O(\lg n)$ сравнений строк. После добавления индекса, указывающего общую длину префикса всех граничных суффиксов, для любого запроса потребуется только $\lg n + |q|$ *сравнений символов*, поскольку появляется возможность определять, какую позицию символа нужно проверить следующим при двоичном поиске. Например, если нижней границей диапазона поиска является слово `cowabunga`, а верхней — `cowslip`, то все промежуточные ключи будут совпадать по первым трем символам, и тогда сравнивать с подстрокой $q$ нужно только четвертый символ любого промежуточного ключа.

На практике скорость поиска в суффиксных массивах обычно такая же, как в суффиксных деревьях или даже выше. Кроме этого, суффиксные массивы требуют меньше — обычно в четыре раза — памяти, чем суффиксные деревья. Каждый суффикс представляется в них уникальной начальной позицией (от 1 до $n$) и может считываться по мере надобности с использованием одной эталонной копии входной строки.

Но для эффективного создания суффиксных массивов нужно соблюдать определенную осторожность, т. к. сортируемые строки содержат $O(n^2)$ символов. Вот одно из решений: сначала создать суффиксное дерево, после чего выполнить симметричный обход этого дерева, чтобы прочитать строки в отсортированном порядке! Но последние достижения в этой области позволяют создавать эффективные по времени и памяти алгоритмы для построения суффиксных массивов напрямую.

### Реализации

В настоящее время существует большое количество реализаций суффиксных массивов. По сути, все последние алгоритмы для построения суффиксных массивов за линейное время были реализованы, а их сравнительная производительность измерена (см. [PST07]). Превосходная реализация суффиксных массивов на языке C описывается в книге [SS07]. Загрузить эту реализацию можно с веб-сайта **https://bibiserv.cebitec.uni-bielefeld.de/bpr/**.

А на веб-сайте Pizza&Chili Corpus (**http://pizzachili.dcc.uchile.cl/**) представлены восемь разных реализаций на C/C++ для сжатых текстовых индексов. Эти структуры данных позволяют значительно минимизировать расход памяти за счет чрезвычайно эффективного сжатия строки входа. При этом они обеспечивают отличное время исполнения запросов.

Вполне доступны и реализации суффиксных деревьев. Проект с открытым кодом BioJava (**http://www.biojava.org**), предоставляющий инфраструктуру Java для обработки биологических данных, содержит класс `SuffixTree`. Реализацию алгоритма Укконена на языке C (называемую Libstree) можно загрузить с веб-сайта **http://www.icir.org/christian/libstree/**. Программы на языке C коллекции strmat реализуют алгоритмы для точного сравнения шаблонов, включая реализацию суффиксных деревьев (см. [Gus97]). Эту коллекцию можно загрузить с веб-сайта **http://web.cs.ucdavis.edu/~gusfield/strmat.html**.

#### *Примечания*

Нагруженные деревья (trie) были впервые предложены Е. Фредкином (E. Fredkin) в журнале [Fre62]. В книге [GBY91] представлен обзор основных структур данных нагруженных деревьев, сопровождаемый многочисленными ссылками.

Эффективные алгоритмы для построения суффиксных деревьев были созданы Вейнером (Weiner), МакКрейтом (McCreight) и Укконеном (Ukkonen) (см. [Wei73], [McC76] и [Ukk92] соответственно). Хорошие описания этих алгоритмов даны в журнале [CR03] и в книге [Gus97]. А в журнале [ACFC+16] излагается интересная, протяженностью в сорок лет, история суффиксных деревьев.

Суффиксные массивы были изобретены Манбером (Manber) и Майерсом (Myers) (см. [MM93]), хотя аналогичная идея Гоннета (Gonnet) и Баеза-Йейтса (Baeza-Yates) была высказана в [GBY91]. Алгоритмы построения суффиксных массивов с линейным временем исполнения были разработаны тремя независимыми командами в 2003 году (см. [KSPP03], [KA03] и [KSB06]), и с тех пор эта область успешно развивается. Обзор всех этих разработок представлен в [PST07].

Результатом последних работ является создание сжатых полнотекстовых индексов, которые позволяют реализовать практически все возможности суффиксных деревьев и массивов в структуре данных, размер которой пропорционален размеру *сжатой* текстовой строки. Обзор этих важных структур данных приведен в [MN07].

Возможности суффиксных деревьев можно расширить, используя структуру данных для вычисления *наименьшего общего предшественника* любой пары узлов $(x, y)$ за постоянное время после предварительной обработки дерева за линейное время. Первоначально эта структура данных была предложена Харелом (Harel) и Тарьяном (Tarjan) (см. [HT84]), после чего упрощена сначала Шибером (Schieber) и Вишкиным (Vishkin) (см. [SV88]), а потом Бендером (Bender) и Фараком (Farach) (см. [BF00]). Ее описание можно найти, например, в [Gus97]. Наименьший общий предшественник двух узлов суффиксного дерева или нагруженного дерева определяет узел, представляющий самый длинный общий префикс двух ассоциированных строк. Удивительно, что осуществление таких запросов происходит за постоянное время, и этот факт делает их пригодными в качестве компонентов многих других алгоритмов.

### Родственные задачи

Поиск подстрок (см. *разд. 21.3*), сжатие текста (см. *разд. 21.5*), поиск самой длинной общей подстроки (см. *разд. 21.8*).

## 15.4. Графы

**Вход.** Граф $G$.

**Задача.** Представить граф $G$ посредством гибкой и эффективной структуры данных (рис. 15.5).

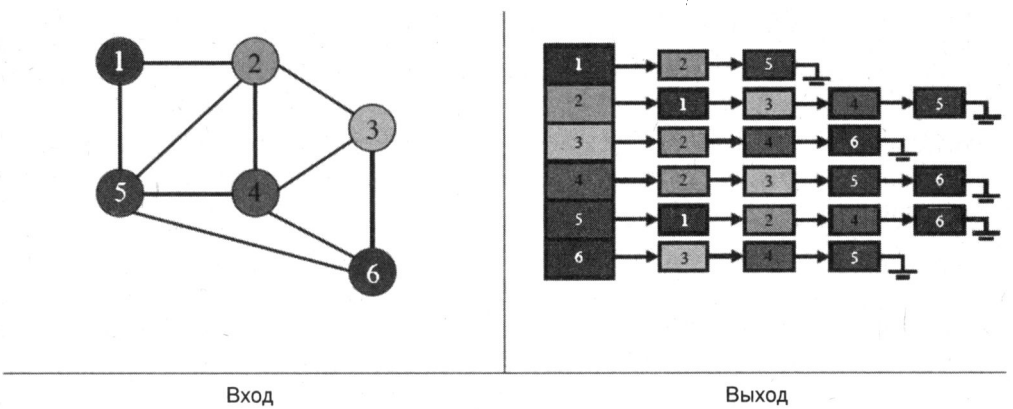



**Рис. 15.5.** Варианты представления графа

### Обсуждение

Двумя основными структурами данных для представления графов являются *матрицы смежности* и *списки смежности*. Подробное описание обеих этих структур данных вместе с реализацией списков смежности на языке *C* дано в *разд. 7.2*. Для большинства задач наиболее подходящей структурой данных являются списки смежности.

Выбирая структуру данных, ответьте на несколько вопросов.

- *Каким будет размер графа?*

  Сколько вершин и ребер будет иметь граф как в типичном, так и в наихудшем случае? Для графов с 1000 вершин требуются матрицы смежности из 1 000 000 элемен-

тов, что в пределах реального. Тем не менее матрицы смежности подходят только для представления небольших или очень плотных графов.

- *Какой будет плотность графа?*

  Если граф очень плотный, т. е. его ребра определены для большей части возможных пар вершин, то, пожалуй, у вас нет особой необходимости использовать списки смежности. В любом случае объем требуемой памяти составит $\Theta(n^2)$, поскольку благодаря отсутствию указателей матрицы смежности для полных графов будут иметь меньший размер.

- *Какой алгоритм применять?*

  В некоторых алгоритмах лучше использовать матрицы смежности (например, в алгоритме поиска кратчайшего пути между всеми парами вершин), но для большинства алгоритмов, основанных на обходе в глубину, предпочитаются списки смежности. Матрицы смежности имеют явное преимущество в алгоритмах, в которых многократно выясняется наличие конкретного ребра в графе. Впрочем, большинство алгоритмов на графах можно разработать так, чтобы подобные запросы в них были исключены. И если это единственные операции, которые должны выполняться, то будет лучше использовать хеш-таблицу ребер.

- *Будет ли граф модифицироваться при выполнении вычислений?*

  Если после создания графа в него не будут вставляться и/или удаляться ребра, то можно использовать эффективные реализации статических графов. В действительности гораздо чаще, чем топология графа, модифицируются *атрибуты* его ребер или вершин: размер, вес, метка или цвет. Атрибуты лучше всего обрабатывать с помощью дополнительных полей в записях ребер или вершин в списках смежности.

- *Будет ли ваш граф перманентной онлайновой структурой?*

  Структуры данных и базы данных — это две разные сущности. Базы данных используются для поддержки коммерческих приложений, которые должные обеспечивать постоянный доступ к большим объемам данных. Я надеюсь, что Facebook не хранит свои графы дружеских отношений в находящемся в памяти списке смежностей. Для онлайнового представления сетей устойчивым образом хорошо подходят базы данных графов — например, Neo4j.

Разработать универсальный программный тип графа весьма непросто. Поэтому я рекомендую проверить наличие уже существующей реализации (главным образом в библиотеке LEDA), прежде чем создавать свою собственную. Обратите внимание, что преобразование между матрицей смежности и списком смежности занимает линейное время по отношению к размеру большей структуры данных. Маловероятно, что это преобразование окажется узким местом в каком-либо приложении, поэтому не исключено использование обеих структур данных, если имеется достаточно памяти. Обычно в этом нет необходимости, но такое решение может оказаться самым простым, если не совсем понятно, какую структуру выбрать.

*Планарными* называются такие графы, которые можно начертить на плоскости без пересечения ребер. Возникающие во многих приложениях графы являются планарными по определению — например, карты стран. Другие графы — например, деревья —

планарны сами по себе. Планарные графы всегда разреженные, т. к. любой планарный граф с $n$ вершинами может иметь самое большее $3n - 6$ ребер. Поэтому для представления этих графов следует использовать списки смежности. Если планарное представление графа (или *укладка*) играет важную роль в решаемой задаче, то планарный граф лучше всего представлять геометрическим способом. Информацию об алгоритмах создания укладок (embeddings) графов см. в *разд. 18.12*.

*Гиперграфами* называются обобщенные графы, в которых каждое ребро может связывать любое количество вершин. Допустим, что мы хотим представить в виде графа членство конгрессменов в разных комитетах. Этот граф и будет гиперграфом, каждая вершина которого представляет отдельного конгрессмена, а каждое гиперребро, соединяющее несколько вершин, представляет один комитет. Такие произвольные наборы подмножеств некоторого множества естественно рассматривать в виде гиперграфов.

При работе с гиперграфами используются следующие базовые структуры данных:

- *матрицы инцидентности*, которые аналогичны матрицам смежности. Для них требуется объем памяти размером $n \times m$, где $m$ — количество гиперребер. Каждая строка такой матрицы соответствует вершине, а каждый столбец — ребру. Значение ячейки $M[i, j]$ будет ненулевым только в том случае, если вершина $i$ входит в ребро $j$. Каждый столбец матрицы инцидентности традиционных графов содержит две ненулевые ячейки. Количество ненулевых позиций в каждой строке зависит от степени каждой вершины;
- *двудольные структуры инцидентности*, которые аналогичны спискам смежности и, следовательно, больше подходят для использования с разреженными гиперграфами. В структуре инцидентности создается вершина, связанная с каждым ребром и вершиной гиперграфа. Для представления такой структуры инцидентности обычно должны использоваться списки смежности. Естественным способом визуального представления гиперграфа является ассоциированный двудольный граф.

Эффективно представить очень большой граф довольно трудно. Тем не менее графы, содержащие миллионы ребер и вершин, использовались для решения интересных задач. В первую очередь следует сделать структуру данных как можно более экономной, упаковав матрицу смежности в битовый вектор (см. *разд. 15.5*) или удалив ненужные указатели из представления списка смежности. Например, в статических графах (не поддерживающих вставку или удаление ребер) каждый список ребер можно заменить упакованным массивом идентификаторов вершин, что позволит избавиться от указателей и потенциально сэкономить половину памяти.

В случае чрезвычайно больших графов может понадобиться иерархическое представление, в котором группы вершин собираются в подграфы, сжимаемые в одну вершину. Такие иерархические декомпозиции можно создавать двумя способами. По первому способу граф разбивается на компоненты естественным или специфическим для приложения способом. Например, сеть дорог и населенных пунктов имеет естественную декомпозицию — разделение карты на населенные пункты, районы и области. Альтернативно можно выполнить алгоритм разбиения графа, рассматриваемый в *разд. 19.6*. Для NP-полной задачи естественная декомпозиция, скорее всего, даст более приемлемый результат, чем наивный эвристический алгоритм. Если же граф действительно

очень большой, вы не сможете позволить себе его алгоритмическое разбиение. Поэтому, прежде чем предпринимать такие решительные действия, убедитесь, что стандартные структуры данных неприменимы к вашей задаче.

### Реализации

Самую лучшую на сегодня реализацию структур данных для представления графов на языке C++ содержит коммерческая библиотека LEDA (см. *разд. 22.1.1*). Изучите представленные в ней методы для работы с графами, чтобы увидеть, как правильный уровень абстракции облегчает задачу реализации алгоритмов.

Более доступной является библиотека Boost Graph Library (см. [SLL02]), которую можно загрузить с веб-сайта **http://www.boost.org/libs/graph**. Библиотека включает реализации списков смежности, матриц смежности и списков ребер, а также неплохую коллекцию базовых алгоритмов для решения задач на графах. Ее интерфейс и компоненты являются обобщенными в том же смысле, что и у библиотеки STL на языке C++.

Также широко используется база данных графов Neo4j (**https://neo4j.com/**), где «j» означает «Java». Примеры графовых алгоритмов с использованием этой базы данных приводятся в [NH19]. Библиотека графов JUNG (**http://jung.sourceforge.net**) особенно популярна среди разработчиков социальных сетей. Библиотека JGraphT (**https://jgrapht.org/**) представляет собой более позднюю разработку с аналогичной функциональностью.

Программа Stanford GraphBase (см. *разд. 22.1.7*) предоставляет простую, но гибкую структуру данных для реализации графов, разработанную с использованием методологии CWEB. Поучительно рассмотреть, что Кнут реализовал (и что не реализовал) в этой базовой структуре данных, хотя в качестве основы для разработок я бы рекомендовал другие реализации.

Среди всех реализаций типов графов на языке C я (субъективно) предпочитаю библиотеки из этой книги, а также из моей книги «Programming Challenges» (см. [SR03]). Подробности см. в *разд. 22.1.9*. Простые структуры данных для представления графов на языке программирования пакета Mathematica, а также библиотека алгоритмов и процедур вывода имеются в библиотеке Combinatorica (см. [PS03]). Подробности см. в *разд. 22.1.8*.

#### *Примечания*

Преимущества списков смежности для работы с графами становятся очевидными при использовании алгоритмов Хопкрофта (Hopcroft) и Тарьяна (Tarjan) с линейным временем исполнения (см. [HT73b], [Tar72]). Базовые структуры данных списков и матриц смежности рассматриваются практически во всех книгах по алгоритмам или структурам данных, включая книги [CLRS09], [AHU83] и [Tar83]. Гиперграфы обсуждаются в книге [Ber89].

Эффективность применения статических графов была показана Неером (Naher) и Злотовским (Zlotowski) в [NZ02], которым удалось повысить скорость исполнения некоторых алгоритмов из библиотеки LEDA в четыре раза, просто используя более компактную структуру для представления графов.

Матричные представления графов могут использовать возможности линейной алгебры для решения задач в диапазоне от нахождения кратчайшего пути до разбиения графов. В [Bap10] представлены пирамиды Лапласа и другие матричные структуры. Интересным

вопросом является минимизация количества битов, необходимых для представления произвольных графов, состоящих из $n$ вершин, особенно если требуется эффективное исполнение определенных операций. Эта тема обсуждается в книге [vL90b].

Динамические алгоритмы решения задач на графах (см. [EGI98]) поддерживают быстрый доступ к инвариантам (таким как минимальное остовное дерево) при вставке или удалении ребер. Общим подходом к созданию динамических алгоритмов является *разрежение* (sparsification) (см. [EGIN97]). Путь в область динамических графовых алгоритмов проложил будущий сценарист ТВ-шоу «The Simpsons» Джефф Уэстбрук (Jeff Westbrook) своей книгой «Algorithms and data structures for dynamic graph problems» («Алгоритмы и структуры данных для динамических задач на графах») [Wes89].

Иерархически определяемые графы часто возникают в задачах разработки микросхем со сверхвысоким уровнем интеграции, т. к. их разработчики активно используют библиотеки схемных элементов (см. [Len90]). Алгоритмы, специфичные для иерархически определяемых графов, включают в себя проверку на планарность (см. [Len89]), проверку на связность (см. [LW88]) и построение минимальных остовных деревьев (см. [Len87a]).

### Родственные задачи

Структуры данных множеств (см. *разд. 15.5*), разделение графа (см. *разд. 19.6*).

## 15.5. Множества

**Вход.** Универсальное множество элементов $U = \{u_1, ..., u_n\}$, по которому определена коллекция подмножеств $S = \{S_1, ..., S_m\}$.

**Задача.** Представить каждое подмножество таким образом, чтобы можно было эффективно проверять вхождение элемента $u_i$ в подмножество $S_j$, вычислять объединение или пересечение подмножеств $S_i$ и $S_j$, вставлять или удалять члены коллекции $S$ (рис. 15.6).

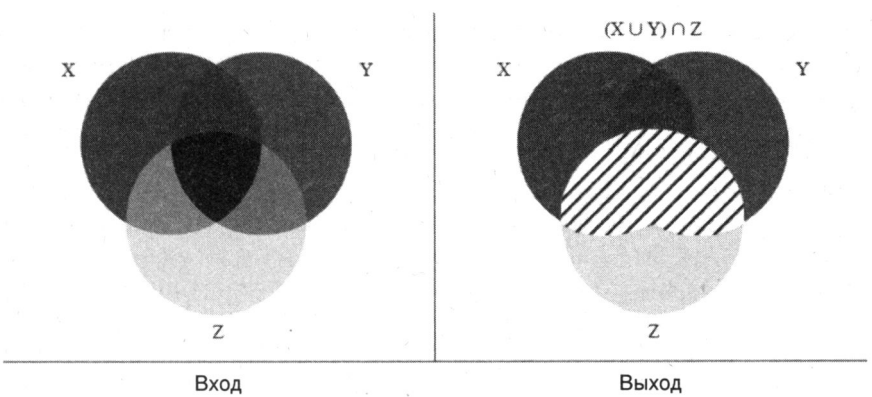



**Рис. 15.6.** Множества X, Y, Z

### Обсуждение

В математике *множеством* называется неупорядоченная коллекция объектов из фиксированного универсального множества. Но обычно удобно представлять множества

в едином *канонически упорядоченном виде*, обычно отсортированном, чтобы ускорить или упростить разные операции. Упорядоченное множество превращает задачу поиска объединения или пересечения двух подмножеств в операцию с линейным временем исполнения — мы просто перебираем элементы слева направо и выясняем, какие отсутствуют. Кроме этого, в упорядоченном множестве поиск элемента занимает сублинейное время. Однако распечатка элементов канонически упорядоченного множества парадоксальным образом свидетельствует о том, что порядок в действительности не имеет большого значения.

Множества отличаются от словарей или строк. Коллекцию объектов, составленную *не* из универсального множества постоянного размера, лучше всего рассматривать как *словарь* (см. *разд. 15.1*). Строки являются структурами, в которых порядок элементов имеет значение, — т. е. строка $\{A, B, C\}$ не равна строке $\{B, C, A\}$. Структуры данных и алгоритмы для работы со строками рассматриваются в *разд. 15.3* и *в главе 21* соответственно.

В *мультимножествах* допустимо вхождение одного и того же элемента более одного раза. Структуры данных для множеств обычно можно расширить на мультимножества, поддерживая поле счетчика или связный список одинаковых вхождений каждого элемента мультимножества.

Когда каждое подмножество содержит ровно два элемента, то эти подмножества могут рассматриваться как ребра графа, вершины которого представляют универсальное множество. Систему подмножеств без ограничений на количество ее элементов можно воспринимать как *гипергpаф*. Вы должны задать себе вопрос, существует ли для вашей задачи аналогия из теории графов — такая как поиск связных компонентов или кратчайшего пути в графе.

Приведем основные структуры данных для представления произвольных подмножеств:

- *битовые векторы.*

  Посредством битового вектора, или массива из $n$ битов, можно представить любое подмножество $S$ из универсального множества $U$, содержащее $n$ элементов. Если $i \in S$, то $i$-му биту присваивается значение 1, а в противном случае — 0. Так как для представления каждого элемента требуется только один бит, то битовые векторы позволяют экономить память даже при очень больших значениях $|U|$. Для вставки или удаления элемента значение соответствующего бита просто меняется на противоположное. Пересечения и объединения вычисляются выполнением логических операций И и ИЛИ. Основным недостатком битового вектора является его неудовлетворительная производительность на разреженных подмножествах. Например, чтобы явно идентифицировать все члены разреженного (или даже пустого) подмножества $S$, требуется время $O(n)$;

- *контейнеры или словари.*

  Подмножество $S$ можно представить посредством связного списка, массива или словаря, содержащего элементы подмножества $S$. Для такой структуры данных не требуется идея постоянного универсального множества. Для разреженных подмножеств словари могут обеспечить бóльшую экономию памяти и времени, чем битовые векторы, к тому же с ними легче работать. Для эффективного выполнения опе-

раций объединения и пересечения следует поддерживать элементы каждого подмножества в отсортированном порядке, что позволяет идентифицировать все дубликаты. объединив оба подмножества за линейное время;

- *фильтры Блума.*

  При отсутствии фиксированного универсального множества битовый вектор можно эмулировать, хешируя каждый элемент подмножества в целое число в диапазоне от 0 до $n - 1$ и устанавливая соответствующий бит. Таким образом, если $e \in S$, то биту $H(e)$ будет присвоено значение 1. Однако при использовании этой схемы могут возникнуть коллизии, поскольку разные ключи могут быть хешированы в одно и то же целое число.

  Чтобы снизить уровень ошибок, в фильтрах Блума используется несколько (скажем, $k$) разных хеш-функций $H_1$, ..., $H_k$, и после вставки ключа $e$ всем $k$ битам $H_i(e)$ присваивается значение 1. Теперь элемент $e$ может быть членом $S$ только в том случае, когда значения всех битов $k$ равны 1. Таким образом, увеличивая количество хеш-функций и размер таблицы, можно снизить вероятность ошибки.

  Эта структура, основанная на хешировании, намного экономичнее словарей в смысле расхода памяти. Она годится для приложений, работающих на статических подмножествах и допускающих небольшую вероятность ошибки. Таких приложений не так уж мало. Например, при проверке правописания не произойдет большой трагедии, если программа иногда пропустит ошибочно написанное слово. Фильтры Блума рассматриваются более подробно в *разд. 6.4*.

Многие приложения работают с попарно непересекающимися коллекциями подмножеств, в которых каждый отдельный элемент является членом только одного подмножества. В качестве примера рассмотрим задачу представления компонентов связности графа или принадлежности политика к партии. Здесь каждая вершина (политик) является членом только одного компонента (партии). Такая система подмножеств называется *разбиением множества*. Алгоритмы генерирования разбиений множества рассматриваются в *разд. 17.6*.

Важным моментом при работе с такими структурами данных является сопровождение изменений на протяжении времени, когда в компонент добавляются или удаляются ребра (в партию вступают новые члены или старые выбывают из нее). Типичные вопросы, возникающие при модифицировании множества путем изменения одного элемента, слияния или объединения двух элементов или же разбиения множества на части, сводятся к двум: «В каком множестве находится определенный элемент?» и «Находятся ли оба элемента в одном и том же множестве?».

Нам понадобятся следующие структуры данных:

- *коллекция контейнеров.*

  Представление каждого подмножества в его собственном контейнере или словаре обеспечивает быстрый доступ ко всем их элементам, что облегчает выполнение операций объединения и пересечения. Но операция проверки членства является дорогой, поскольку требует отдельного поиска в каждой структуре данных до тех пор, пока не будет найден искомый элемент;

- *обобщенный битовый вектор.*

  Пусть *i*-й элемент массива содержит номер/имя содержащего его подмножества. Запросы, идентифицирующие множество, и модификации отдельных элементов можно выполнять за постоянное время. Но для объединения двух подмножеств может потребоваться время, пропорциональное размеру универсального множества, т. к. необходимо идентифицировать каждый элемент в этих двух подмножествах и изменить его имя;

- *словарь с атрибутом подмножества.*

  Подобным образом каждый элемент в двоичном дереве допускается связать с полем, в котором хранится имя его подмножества. Запросы для идентификации множества и модифицирования одного элемента можно выполнять за время, требуемое для выполнения поиска в словаре. Однако операции объединения и пересечения попрежнему выполняются медленно. Необходимость эффективно исполнять операции объединения заставляет нас использовать структуру данных «объединение-поиск» (см. далее);

- *структура данных «объединение-поиск».*

  Подмножество здесь представляется посредством корневого дерева, в котором каждый узел указывает вместо своего потомка на своего предшественника. Именем каждого подмножества служит имя корневого элемента. Чтобы определить имя подмножества для того или иного элемента, просто перемещаемся от одного указателя на родительский элемент на другой, пока не дойдем до корня. Операция объединения двух подмножеств также не представляет сложностей. Мы просто присваиваем корню одного из деревьев указатель на другое, после чего *все* элементы получают общий корень и, следовательно, одно и то же имя подмножества.

  В этом случае детали реализации оказывают большое влияние на асимптотическую оценку производительности. Если при выполнении операции слияния всегда выбирать большее (более высокое) дерево в качестве корневого, то в результате гарантированно получатся деревья логарифмической высоты (см. *разд. 8.1.3*). Сокращение проложенного пути после каждого нахождения элемента при снабжении всех узлов в этом пути явными указателями на корень называется *сжатием пути* и сводит высоту дерева к почти константной. Структура данных «объединение-поиск» — простая и быстро работающая, и с ней должен быть знаком каждый программист.

### Реализации

Современные языки программирования включают библиотеки полных и эффективных реализаций множеств. Библиотека стандартных шаблонов STL языка C++ содержит контейнеры множеств (`set`) и мультимножеств (`multiset`). Библиотека LEDA (см. *разд. 22.1.1*) содержит эффективные структуры данных словарей, разреженные массивы и структуры данных «объединение-поиск» для работы с разбиениями множеств — все на языке C++. Стандартный пакет утилит java.util включает в себя библиотеку Java Collections (JC), которая содержит контейнеры `HashSet` и `TreeSet`.

Реализация структуры данных «объединение-поиск» лежит в основе любой реализации алгоритма Крускала для построения минимального остовного дерева. Поэтому предпо-

лагается, что все библиотеки графов, упоминаемые в *разд. 15.4*, содержат эту реализацию. Реализации минимальных остовных деревьев рассматриваются в *разд. 18.3*.

Система компьютерной алгебры REDUCE (**http://www.reduce-algebra.com**) содержит пакет SETS, который поддерживает теоретико-множественные операции как над явными, так и над неявными (символическими) множествами. Другие системы компьютерной алгебры могут иметь аналогичную функциональность.

#### Примечания

Оптимальные алгоритмы для таких операций над множествами, как пересечение и объединение, были представлены в публикации [Rei72]. В книге [Ram05] прекрасно описаны структуры данных для выполнения ряда разных операций над множествами. Квалифицированно выполненный обзор фильтров Блума приведен в журнале [BM05], а результаты последних экспериментов в этой области изложены в докладе [PSS07]. *Кукушкин фильтр* (см. [FAKM14]) представляет собой улучшенный вариант фильтра Блума, обеспечивающий повышенную эффективность по времени исполнения и в использовании памяти, а также поддерживающий удаления.

Некоторые структуры данных сбалансированных деревьев поддерживают операции `merge`, `meld`, `link` и `cut`, что позволяет быстро выполнять операции объединения над непересекающимися подмножествами. Хорошее описание таких структур можно найти в книге [Tar83]. Джекобсон (Jacobson) улучшил структуру данных битового вектора для эффективной поддержки операции выбора (поиска $i$-го установленного бита) как по времени, так и по памяти.

Обзор структур данных для объединения непересекающихся множеств представлен в работе [GI91]. Верхний предел производительности, равный $O(m\alpha(m, n))$, для $m$ операций поиска и объединения в $n$-элементном множестве вычислен Тарьяном (Tarjan) (см. [Tar75]). Им же найден и соответствующий нижний предел в ограниченной модели вычислений (см. [Tar79]). Так как обратная функция Аккермана $\alpha(m, n)$ возрастает крайне медленно, то производительность приближается к линейной. Интересная связь между наихудшим случаем исполнения операций поиска и объединения и длиной последовательности Дэвенпорта — Шинцеля (Davenport — Schinzel) — комбинаторной структуры, рассматриваемой в вычислительной геометрии — представлена в книге [SA95].

*Степенным множеством* (power set) множества $S$ называется коллекция всех $2^{|S|}$ подмножеств множества $S$. Явная манипуляция степенными множествами быстро становится трудной вследствие их большого размера. Для нетривиальных вычислений степенные множества требуется представлять неявно в символической форме. Информацию об алгоритмах работы с символическими представлениями степенных множеств и результатов вычислительных экспериментов с ними можно найти в [BCGR92].

### Родственные задачи

Генерирование подмножеств (см. *разд. 17.5*), генерирование разбиений (см. *разд. 17.6*), вершинное покрытие (см. *разд. 21.1*), минимальное остовное дерево (см. *разд. 18.3*).

## 15.6. Kd-деревья

**Вход.** Множество $S$, состоящее из $n$ точек (или более сложных геометрических объектов) в $k$-мерном пространстве.

**Задача.** Создать дерево, которое делит пространство полуплоскостями таким образом, что каждый объект содержится в своей собственной $k$-мерной прямоугольной области (рис. 15.7).

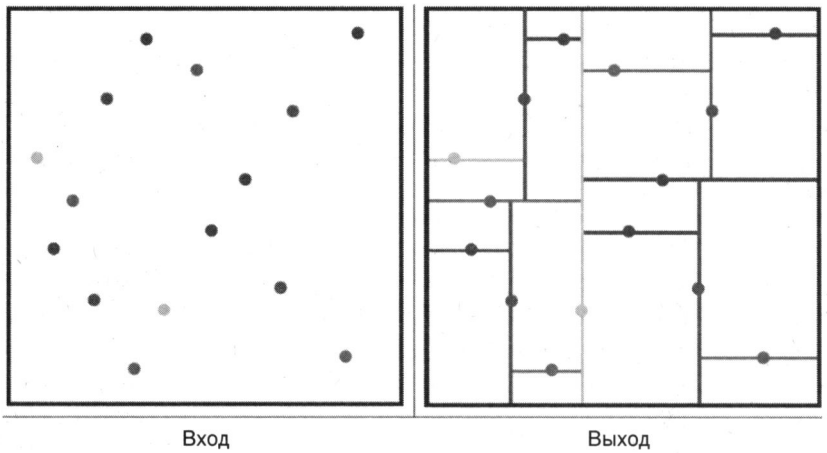



**Рис. 15.7.** Создание kd-дерева

### Обсуждение

Kd-деревья и связанные с ними пространственные структуры данных иерархически разбивают пространство на небольшое количество ячеек, каждая из которых содержит всего лишь несколько представителей входного множества точек. Это обеспечивает быстрый доступ к любому объекту по его позиции. Мы просто проходим вниз по иерархической структуре, пока не дойдем до наименьшей ячейки, содержащей требуемый элемент, после чего перебираем все элементы ячейки, пока не найдем тот, который нам нужен.

Обычно алгоритмы создают kd-деревья путем разбиения множества точек. Каждый узел дерева определяется плоскостью, проходящей через одно из измерений. В идеале, эта плоскость разбивает множество точек на равные правое и левое (или верхнее и нижнее) подмножества. Полученные подмножества опять разбиваются на равные половины плоскостями, проходящими через другое измерение. Процесс прекращается после $\lg n$ разбиений, при этом каждая точка оказывается в своей собственной листовой ячейке.

Секущие плоскости вдоль любого пути от корня к другому узлу определяют уникальную прямоугольную область пространства. Каждая следующая плоскость разделяет эту область на две меньшие. Каждая прямоугольная область определяется $2k$ плоскостями, где $k$ — количество измерений[1]. По мере продвижения вниз по дереву мы продолжаем ограничивать интересующую нас область этими полупространствами.

Тип kd-дерева определяется выбором секущей плоскости. Возможны следующие варианты:

[1] Попутно отметим, что «kd» в слове «kd-дерево» — это сокращение от «k-dimensional», «k-мерное».

- *циклический проход по измерениям.*

  Сначала разбивается измерение $d_1$, потом $d_2$, ..., $d_k$, после чего мы возвращаемся к $d_1$;

- *сечение вдоль самого большого измерения.*

  Выбираем такое измерение, чтобы форма получившихся в результате его сечения областей была как можно ближе к квадратной. Разбиение области точек пополам не обязательно означает, что секущая плоскость проходит точно посередине прямоугольной области, поскольку все точки могут лежать у одного края подлежащей сечению области;

- *квадрадеревья и октадеревья.*

  Вместо сечения одной плоскостью используются плоскости, параллельные всем координатным осям, которые проходят через ту или иную точку сечения. Это означает, что в двух измерениях создаются четыре дочерние ячейки (квадрадерево), а в трех измерениях — восемь (октадерево). Использование квадрадеревьев особенно популярно при работе с видеоданными, когда листья дерева таковы, что все пикселы в этих областях имеют одинаковый цвет;

- *RP-деревья*[2].

  Здесь секущая плоскость узла определяется случайным наклоном или направлением (или, в общем, это ($d$ – 1)-размерная плоскость, проходящая через начало координат). Мы определяем линию или плоскость, перпендикулярную этому направлению, которая делит область точек на две равные половины, — такая линия существует, и ее можно легко найти. После этого процесс повторяется. Полученные деревья должны иметь логарифмическую высоту, но при этом могут следовать точкам лучше, чем kd-деревья, когда низкоразмерные структуры встречаются в данных более высокой размерности;

- *BSP-деревья*[3]. При двоичном разбиении пространства используются секущие плоскости общего вида (т. е. не обязательно параллельные координатным осям), чтобы разделить пространство на ячейки таким образом, чтобы каждая ячейка содержала только один объект (скажем, многоугольник). Для некоторых наборов объектов такое разделение невозможно осуществить посредством только секущих плоскостей, параллельных координатным осям. Недостатком этого подхода является то обстоятельство, что получившиеся многогранные ячейки труднее поддаются обработке, чем ячейки прямоугольной формы;

- *Шаровые деревья.*

  Шаровое дерево (ball tree) — это иерархическая структура данных по точкам, в которой каждый узел связан со сферой (шаром), определяемой центром и радиусом. Сфера каждого узла — наименьшая, которая содержит сферы его потомков. В отличие от kd-деревьев, это означает, что сестринские узлы могут пересекаться и не раз-

---

[2] Random Projection Trees — деревья случайных проекций.

[3] Binary Space Partitions — разделы двоичного пространства.

деляют все пространство. Но они очень хорошо подходят для поиска ближайшего узла в высокоразмерных пространствах, где kd-деревья пасуют.

В идеале сечения разделяют поровну как пространство (давая в результате большие области правильной формы), так и набор точек (обеспечивая дерево логарифмической высоты), но для какого-либо экземпляра входа выполнить сечение таким образом может оказаться невозможным. Преимущества ячеек большого размера становятся очевидными во многих следующих применениях kd-деревьев:

- *Определение местоположения точки.*

  Для выявления ячейки, в которой находится целевая точка $q$, выполняется обход дерева, начиная с корня, в процессе которого оба полупространства с каждой стороны секущей плоскости проверяются на наличие в них искомой точки. Повторяя этот процесс на соответствующем дочернем узле, мы спускаемся вниз по дереву, пока не найдем листовую ячейку с искомым элементом $q$. Такой поиск занимает время, пропорциональное высоте дерева. Дополнительную информацию по вопросу определения местоположения точки см. в *разд. 20.7*.

- *Поиск ближайшей точки.*

  Чтобы в множестве $S$ найти точку, ближайшую к искомой точке запроса $q$, выполняется процедура определения местоположения ячейки $c$, содержащей точку $q$. Так как эта ячейка связана с какой-либо точкой $p$, то можно вычислить расстояние $d(p, q)$ от точки $p$ до точки $q$.

  Точка $p$, скорее всего, расположена вблизи точки $q$, но она может быть не абсолютно ближайшей точкой. Почему? Допустим, что точка $q$ находится у правой границы своей ячейки. Тогда точка, ближайшая к точке $q$, может быть расположена слева от этой границы в соседней ячейке. Значит, нам нужно обойти все ячейки, которые находятся от ячейки $q$ на расстоянии, не превышающем $d(p, q)$, и проверить, что ни одна из них не содержит никаких более близких точек. В дереве из крупных ячеек правильной формы проверить придется очень небольшое количество ячеек. Поиск ближайшей точки подробно обсуждается в *разд. 20.5*.

- *Поиск в диапазоне.*

  Как определить, какие точки находятся в целевой области? Начиная с корня дерева, проверяем целевую область на пересечение с ячейкой (или на вхождение в нее этой ячейки), определяющей текущий узел. При положительном результате проверки выполняем такую же проверку на потомках. В противном случае никакая из листовых ячеек ниже этого узла уже не будет нас интересовать. Так быстро отсекаются не имеющие отношения к делу области пространства. Поиск точки в указанном диапазоне подробно обсуждается в *разд. 20.6*.

- *Поиск по частичному ключу.*

  Допустим, что нам нужно найти точку $p$ в множестве $S$, но мы не располагаем подробной информацией об этой точке. Предположим, мы ищем в kd-дереве, имеющем измерения для возраста, роста и веса, узел, представляющий человека в возрасте 59 лет ростом 175 сантиметров и с неизвестным весом. Начиная путь от корня, мы можем найти подходящий потомок по всем измерениям, кроме веса. Чтобы иметь уверенность, что найдена нужная точка, мы должны искать *оба потомка* этих узлов.

Чем больше полей (измерений) известно, тем лучше, но такой поиск точки по частичной информации о ней может пройти значительно быстрее, чем сравнение всех точек с искомым ключом.

Kd-деревья лучше всего годятся для приложений с умеренным количеством измерений — в диапазоне от 2 до 20. С увеличением количества измерений они теряют эффективность в основном из-за того, что объем единичного шара в $k$-мерном пространстве уменьшается экспоненциально, в отличие от объема единичного куба. Следовательно, в приложениях, связанных, например, с поиском ближайшей точки, экспоненциально возрастает количество ячеек, проверяемых в пределах заданного радиуса с центром в точке запроса. Кроме этого, для любой ячейки количество смежных ячеек увеличивается до $2k$ и, в конце концов, становится неуправляемым.

Главный вывод из сказанного состоит в том, что следует избегать работы в многомерных пространствах, возможно, отказываясь от рассмотрения наименее важных измерений.

### Реализации

Программа KDTREE 2 содержит реализации kd-деревьев на языках C++ и FORTRAN 95 для эффективного поиска ближайшего соседа во многих измерениях. Подробности вы найдете по адресу **http://arxiv.org/abs/physics/0408067**.

Деревья шариков включены как часть популярного пакета Python scikit-learn, опять же для поиска ближайшего соседа в многомерных данных.

Шаровые деревья входят в состав популярного пакета Python scikit-learn и позволяют вести поиск ближайшей точки в высокоразмерных данных.

Апплеты Java коллекции Spatial Index Demos (**http://donar.umiacs.umd.edu/quadtree**) демонстрируют многие варианты kd-деревьев. Алгоритмы, реализуемые в апплетах, рассматриваются в книге [Sam06].

Соревнование DIMACS в 1999 году фокусировалось на структурах данных для поиска ближайшего соседа (см. [GJM02]). Наборы данных и коды можно загрузить с веб-сайта **http://dimacs.rutgers.edu/Challenges**.

#### *Примечания*

Самым лучшим справочником по kd-деревьям и другим пространственным структурам данных является книга [Sam06]. В ней подробно рассмотрены все основные (и многие второстепенные) варианты этих структур. Книга [Sam05] — краткий обзор этих структур. Принято считать, что kd-деревья разработал Дж. Бентли (J. Bentley) (см. [Ben75]), но история их появления туманна, как и у большинства популярных структур данных.

Большое количество измерений ведет к падению производительности пространственных структур данных. Для решения этой проблемы используются среди прочих такие структуры данных, как шаровые деревья (см. [Omo89]) и деревья случайных проекций (см. [DF08]).

Недавно был представлен простой, но мощный метод понижения размерности, заключающийся в отображении многомерных пространств на произвольную гиперплоскость меньшей размерности. Как теоретические, так и экспериментальные результаты (см. [IMS18] и [BM01] соответственно) указывают на то, что этот метод сохраняет расстояния довольно хорошо.

Важным аспектом в области поиска ближайшего соседа в пространственных структурах высокой размерности являются алгоритмы для быстрого поиска точки, которая находится доказуемо близко к точке запроса. Результаты последних экспериментов по нахождению близлежащих точек в пространственных структурах высокой размерности приводятся в [ML14]. Очень популярные и эффективные методы LSH-хеширования излагаются в [AI06]. А в [AMN+98] описывается другой подход, который на основе набора данных создает разреженный взвешенный граф, после чего поиск ближайшего соседа осуществляется с применением «жадного» обхода от произвольной точки по направлению к точке запроса. Ближайшим соседом считается точка, найденная в результате нескольких таких операций поиска, начатых от произвольной точки. Подобные методы являются перспективными для решения других задач в пространствах с большим количеством измерений.

### Родственные задачи

Поиск ближайшего соседа (см. *разд. 20.5*), определение местоположения точки (см. *разд. 20.7*), поиск в диапазоне (см. *разд. 206.6*).

# Численные задачи

Если вам в основном приходится иметь дело с численными задачами, значит, вы читаете не ту книгу. Вам тогда лучше подойдет книга [PFTV07], в которой дается отличный обзор фундаментальных задач в области численных методов, включая линейную алгебру, численное интегрирование, статистику и дифференциальные уравнения. Соответствующие версии этой книги содержат исходный код для всех рассматриваемых в ней алгоритмов на языке C++, FORTRAN и даже Pascal. Комбинаторно-численные задачи, рассматриваемые в этой главе, обсуждаются в ней не очень подробно, но тем не менее о существовании этой книги следует знать. Дополнительную информацию см. на веб-сайте **http://numerical.recipes/.**

В последнее время численные расчеты приобретают все большую важность вследствие развития машинного обучения, которое в значительной мере полагается на линейную алгебру и безусловную оптимизацию. Но обратите внимание на то, что между численными и комбинаторными алгоритмами существуют, по крайней мере, два различия:

- *по вопросам точности.*

  Численные алгоритмы обычно выполняют повторяющиеся вычисления с плавающей точкой, и с каждой операцией накапливается ошибка, пока результаты в конце концов не теряют всякий смысл. Моим любимым примером такого накопления ошибки вычислений является случай с индексом Ванкуверской фондовой биржи, в котором за 22 месяца накопилась столь большая ошибка округления, что индекс уменьшился до значения 574,081, в то время как правильное значение должно было быть 1098,982 (см. [MV99]).

  Существует простой и надежный способ проверки погрешностей округления в численных программах. Для этого программа исполняется на данных как с одинарной, так и с двойной точностью, после чего результаты сравниваются;

- *по наличию библиотек кода.*

  Начиная с 1960-х годов для численных алгоритмов создано большое количество высококачественных библиотек процедур, каких еще нет для комбинаторных алгоритмов. Причин этого несколько, включая:

  - раннее становление языка FORTRAN в качестве стандарта для численных расчетов;
  - независимость модулей численных расчетов от приложений, в которых они применяются;
  - существование крупных научных сообществ, нуждающихся в библиотеках общего назначения для решения численных задач.

Скорее всего, у вас нет никаких причин для самостоятельной реализации алгоритмов для решения любой задачи, описанной в этой главе. Я рекомендую начинать поиски готового кода с библиотеки NetLib (см. *разд. 22.1.4*).

Многие ученые и инженеры имеют идеи об алгоритмах, которые в культурном отношении произошли от простого управления и структур данных численных методов.

Представления многих ученых и инженеров об алгоритмах основаны на простом управлении и несложных структурах данных численных методов. В противоположность им программисты изучают в студенческие годы работу с указателями и применение рекурсии, поэтому они чувствуют себя свободно с более сложными структурами данных, используемыми в комбинаторных алгоритмах. Оба эти сообщества могут и должны учиться друг у друга, поскольку для моделирования многих задач можно применять как численный, так и комбинаторный подход.

Существует множество книг, посвященных численным алгоритмам. В частности, в дополнение к уже упомянутой ранее книге я рекомендую для ознакомления и следующие:

- «Numerical Methods for Engineers» («Численные методы для инженеров») [CC16] — бестселлер среди руководств по численному анализу;
- «The Java Programmer's Guide to Numerical Computing» («Руководство программиста Java по численным расчетам») [Mak02] — хорошо написанное учебное пособие, в котором язык Java вводится в мир численных расчетов. В книгу включен исходный код примеров;
- «Numerical Methods for Scientists and Engineers» («Численные методы для ученых и инженеров») [Ham87] — старое, но не устаревшее пособие содержит четкое и понятное рассмотрение фундаментальных методов численных расчетов;
- «Elementary Numerical Computing with Mathematica» («Элементарные численные расчеты с Mathematica») [SK00] — интересное обсуждение базовых численных методов, в котором нет слишком подробного описания алгоритмов благодаря использованию системы вычислительной алгебры Mathematica. На мой взгляд, это очень хорошая книга;
- «Numerical Mathematics and Computing» («Вычислительная математика и расчеты») [CK12] — традиционный учебник по численному анализу, написанный с использованием языка FORTRAN и включающий обсуждение методов оптимизации и метода Монте-Карло в дополнение к таким традиционным темам, как извлечение корня, численное интегрирование, системы линейных уравнений, сплайны и дифференциальные уравнения;
- «Numerical Methods and Analysis» («Численные методы и анализ») [BT92] — всестороннее и не привязанное ни к какому конкретному языку программирования обсуждение всех стандартных тем, включая параллельные алгоритмы. Это наиболее полное изо всех упомянутых здесь руководств.

# 16.1. Решение системы линейных уравнений

**Вход.** Матрица $A$ размером $m \times n$ и вектор $b$ размером $m \times 1$, совместно представляющие $m$ линейных уравнений с $n$ переменными.

**Задача.** Найти такой вектор $x$, для которого $A \cdot x = b$ (рис. 16.1).

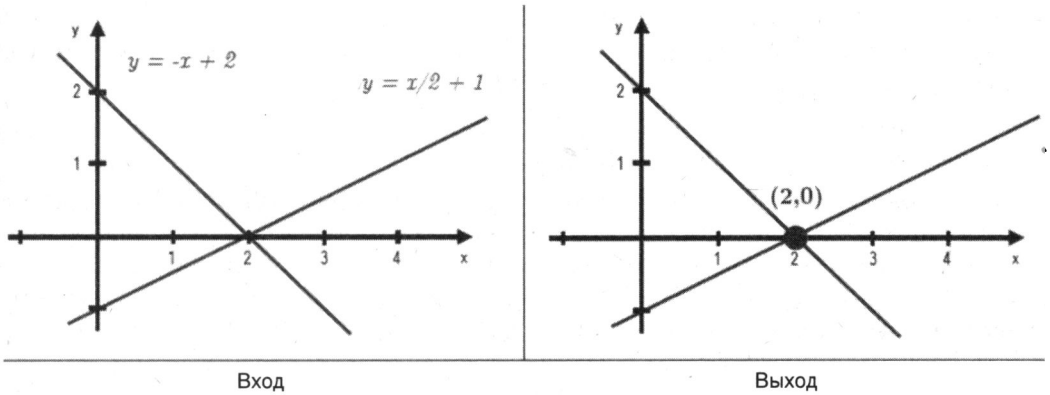



**Рис. 16.1.** Решение системы линейных уравнений

### Обсуждение

Потребность в решении систем линейных уравнений возникает в приблизительно 75% всех научных вычислительных задач (см. [DB74]). Например, применение закона Кирхгофа для анализа электрических схем порождает систему уравнений, решение которой определяет прохождение тока через каждую ветвь схемы. Многие алгоритмы машинного обучения сводятся к решению систем линейных уравнений, включая линейную регрессию и разложение по сингулярным числам матрицы. Даже задача поиска точки пересечения двух или более линий сводится к решению небольшой системы линейных уравнений.

Но не все системы уравнений имеют решения — например, такая система решения не имеет:

$$\begin{cases} 2x + 3y = 5 \\ 2x + 3y = 6 \end{cases}$$

А некоторые системы уравнений, наоборот, имеют несколько решений — например, эта:

$$\begin{cases} 2x + 3y = 5 \\ 4x + 6y = 10 \end{cases}$$

Такие системы уравнений называются *вырожденными*, и их можно распознать, проверив определитель матрицы коэффициентов на равенство нулю.

Задача решения систем линейных уравнений настолько важна в научных и экономических приложениях, что для ее решения существует достаточное количество превосходных программных реализаций. Так что нет никакого смысла создавать собственную реализацию, хотя базовый алгоритм (метод исключения Гаусса) и изучается в старших классах средней школы. Особенно это справедливо в отношении систем с большим количеством неизвестных.

Метод исключения Гаусса основан на том обстоятельстве, что при умножении уравнения на константу его решение не меняется (если $x = y$, то и $2x = 2y$) и при сложении уравнений системы ее решение тоже не меняется (т. е. решение системы уравнений

$x = y$ и $w = z$ такое же, как и решение системы $x = y$ и $x + w = y + z$). При решении методом Гаусса уравнения умножаются на константу и складываются так, чтобы можно было последовательно исключать переменные из уравнений и привести систему к ступенчатому виду.

Временна́я сложность метода Гаусса для системы из $n \times n$ уравнений равна $O(n^3)$, поскольку для того, чтобы исключить $i$-ю (из $n$) переменную, мы добавляем умноженную на константу копию $n$-го члена $i$-й строки к каждому из оставшихся $n - 1$ уравнений. Но в этой задаче константы играют важную роль. Алгоритмы, которые для получения решения выполняют только частичную обработку матрицы коэффициентов, а затем производят обратную замену, используют на 50% меньше операций с плавающей точкой, чем простые алгоритмы.

Прежде чем приступать к решению системы линейных уравнений, вам нужно ответить на следующие вопросы:

- *Влияют ли на решение ошибки округления?*

  Реализация метода Гаусса была бы совсем простым делом, если бы не ошибки округления. Они накапливаются с каждой операцией и вскоре полностью искажают решение, особенно для *почти* вырожденных матриц.

  Чтобы устранить влияние ошибок округления, следует подставлять решение в каждое из первоначальных уравнений и проверять, насколько оно близко к искомому значению. Повысить точность решений систем линейных уравнений можно с помощью *итерационных методов* решения — таких как, например, методы Якоби и Гаусса — Зейделя. Хорошие пакеты для решения систем линейных уравнений должны содержать такие процедуры.

  Ключом к минимизации ошибок округления в решениях методом Гаусса является выбор правильных опорных уравнений и переменных и умножение уравнений на константы для исключения больших множителей. Это особое искусство, и вам следует использовать тщательно разработанные библиотечные процедуры, описанные далее.

- *Какую библиотечную процедуру нужно использовать?*

  Умение выбрать правильный код также является своего рода искусством. Если вы следуете советам из этой книги, то начните с какого-либо неспециализированного пакета для решения систем линейных уравнений. Очень вероятно, что его будет достаточно для ваших задач. Однако вам следует поискать в руководстве пользователя эффективные процедуры для решения особых типов систем линейных уравнений. Если окажется, что ваша матрица принадлежит к одному из специальных типов, поддерживаемых пакетом, то время ее решения может быть уменьшено с кубического до квадратичного или даже линейного.

- *Является ли решаемая система разреженной?*

  Если в матрице $A$ имеется небольшое количество ненулевых элементов, то матрица считается *разреженной* (sparse) и нам повезло. Если эти несколько ненулевых элементов сосредоточены возле диагонали, то матрица считается *ленточной* (banded) и нам повезло еще больше. Алгоритмы для уменьшения ширины ленты матрицы рассматриваются в *разд. 16.2*. Для решения можно воспользоваться многими дру-

гими стандартными образцами разреженных матриц, подробности о которых можно узнать в руководстве пользователя выбранного вами программного пакета или в хорошей книге, целиком посвященной численному анализу.

- *Будет ли использоваться одна и та же матрица коэффициентов для решения нескольких систем?*

  В таких приложениях, как аппроксимация кривой методом наименьших квадратов, уравнение $A \cdot x = b$ часто решается несколько раз с разными векторами $b$. Чтобы облегчить этот процесс, матрицу $A$ можно подвергнуть предварительной обработке. *LU-разложение* — это представление матрицы $A$ с помощью нижней ($L$) и верхней ($U$) треугольных матриц, для которых $L \cdot U = A$. LU-разложение можно использовать для решения уравнения $A \cdot x = b$, поскольку

  $$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b.$$

  Это эффективное решение, т. к. обратная подстановка позволяет решить треугольную систему уравнений за квадратичное время. После выполнения LU-разложения за время $O(n^3)$ решение уравнения $L \cdot y = b$, а потом уравнения $U \cdot x = y$ дает нам решение для $x$ за два шага с временем исполнения каждого $O(n^2)$ вместо одного шага с временем исполнения $O(n^3)$.

Задача решения системы линейных уравнений эквивалентна задаче обращения матрицы, т. к. $Ax = B \leftrightarrow A^{-1}Ax = A^{-1}B$, где $I$ — единичная матрица ($I = A^{-1}A$). Но этого подхода следует избегать, поскольку он в три раза медленнее, чем метод исключения Гаусса. LU-разложение оказывается очень полезным как для обращения матриц, так и для вычисления определителей (см. *разд. 16.4*).

### Реализации

Самой лучшей библиотекой для решения систем линейных уравнений по-видимому является библиотека LAPACK — более поздняя версия библиотеки LINPACK (см. [DMBS79]). Обе эти библиотеки, написанные на языке FORTRAN, являются частью библиотеки NetLib (**https://www.netlib.org/**). Версии библиотеки LAPACK имеются и для других языков, в частности для C — CLAPACK и для C++ — LAPACK++. Для этих процедур имеется интерфейс на языке C++, Template Numerical Toolkits, который можно загрузить с веб-сайта **http://math.nist.gov/tnt**.

Универсальная библиотека для научных расчетов JScience содержит обширный пакет инструментов для линейной алгебры (включая определители). Библиотека JAMA также представляет собой пакет для работы с матрицами на языке Java. Ссылки на эти и другие подобные библиотеки вы найдете на сайте **http://math.nist.gov/javanumerics**.

Еще одним источником руководства и процедур для решения систем линейных уравнений является книга [PFTV07] (**http://numerical.recipes/**). Наиболее убедительная причина для использования этих реализаций вместо бесплатных — это отсутствие у разработчика уверенности при работе с численными методами.

#### Примечания

Стандартным справочником по алгоритмам для работы с системами линейных уравнений является книга [GL96]. Хорошие описания алгоритмов для метода Гаусса и LU-разложения можно найти в книге [CLRS09]. Существует множество пособий по численному

анализу — таких как [BT92], [CK12], [SK00]. Обзор структур данных для систем линейных уравнений представлен в книге [PT05].

Параллельные алгоритмы для систем линейных уравнений обсуждаются в публикациях [Gal90], [GO14], [HNP91] и [KSV97]. Решение систем линейных уравнений является одним из наиболее важных приложений, в которых на практике широко применяются параллельные архитектуры.

Обращение матриц и соответственно решение систем линейных уравнений можно выполнить за время, требуемое для умножения матриц, с помощью алгоритма Штрассена с последующим сведением. Среди хороших описаний равноценных задач можно назвать [AHU74] и [CLRS09].

Для решения систем линейных уравнений размером $n \times n$ было предложено использовать алгоритм HHL для квантовых вычислений с временной сложностью $O(\log n)$ (см. [HHL09]), что дает экспоненциальное ускорение по сравнению с обычными компьютерами. Это было бы серьезным прорывом в этой области, но здесь есть много оговорок, которые подробно рассматриваются в [Aar15].

### Родственные задачи

Умножение матриц (см. *разд. 16.3*), определители и перманенты (см. *разд. 16.4*).

## 16.2. Уменьшение ширины ленты матрицы

**Вход.** Граф $G = (V, E)$, представляющий матрицу $M$ размером $n \times n$.

**Задача.** Найти такую перестановку вершин $p$, которая минимизирует самое длинное ребро, — т. е. минимизирует $\max_{(i,j)} \in E\ |p(i) - p(j)|$ (рис. 16.2).

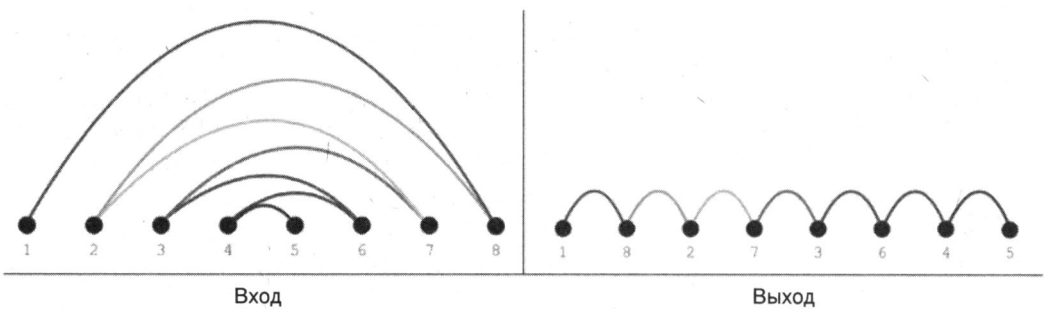



**Рис. 16.2.** Уменьшение ширины ленты матрицы

### Обсуждение

Уменьшение ширины ленты представляет собой незаметную на первый взгляд, но очень важную задачу, как для графов, так и для матриц. Применительно к матрицам, уменьшение ширины ленты заключается в перестановке строк и столбцов разреженной матрицы, чтобы минимизировать расстояние $b$ от центральной диагонали самого дальнего ненулевого элемента. Это важная операция в решении систем линейных уравнений, поскольку на матрицах с лентой шириной $b$ метод Гаусса исполняется за время $O(nb^2)$, что является большим улучшением общего времени исполнения $O(n^3)$ алгоритма, когда $b \ll n$.

Уменьшение ширины ленты в графах проявляется не столь очевидным образом. В качестве одного из примеров задачи уменьшения ширины ленты в графах можно привести задачу упорядочивания $n$ элементов электрической схемы в линию таким образом, чтобы минимизировать длину самого длинного проводника (и таким образом минимизировать временную задержку). В этом случае каждая вершина графа $G$ представляет элемент схемы, а каждое ребро — проводник, соединяющий два элемента. Более общие задачи, такие как компоновка прямоугольных схем, наследуют сложность и эвристические классы линейных версий.

В задаче *уменьшения ширины ленты* требуется упорядочить вершины в линию таким образом, чтобы минимизировать длину самого длинного ребра, но существуют и другие варианты задачи. В задаче *линейного расположения* (linear arrangement) требуется минимизировать сумму длин всех ребер. Эту задачу можно применить для компоновки схем, в которой требуется расположить микросхемы таким образом, чтобы минимизировать общую длину проводников. В задаче *минимизации профиля* требуется минимизировать сумму длин однонаправленных ребер (т. е. для каждой вершины $v$ минимизировать длину самого большого ребра, вторая вершина которого находится слева от вершины $v$).

К сожалению, задача уменьшения ширины ленты во всех ее вариантах является NP-полной. Она остается NP-полной, даже если графом входа является дерево с максимальной степенью вершины, равной 3, что является самым строгим ограничением, которое я когда-либо встречал. Поэтому единственными способами ее решения являются метод полного перебора решений и эвристический подход.

К счастью, эвристические методы специального назначения хорошо изучены, и для наилучших из них имеются высококачественные реализации. В основе этих реализаций лежит обход в ширину, начинающийся с определенной вершины $v$, где эта вершина помещается в самую левую позицию упорядочения. Все вершины на расстоянии 1 от вершины $v$ помещаются непосредственно справа от нее, за ними следуют вершины на расстоянии 2 и т. д., пока не будут обработаны все вершины графа. Популярные эвристические алгоритмы различаются по количеству рассматриваемых начальных вершин и по способу упорядочивания равноудаленных вершин. В последнем случае хорошим методом обычно является выбор самой левой из равнозначных вершин низкой степени.

Реализации наиболее популярных эвристических алгоритмов: Катхилла — Макки (Cuthill — McKee) и Гиббса — Пула — Стокмейера (Gibbs — Poole —Stockmeyer) — рассматриваются далее — в *подразделе «Реализации»*. Время исполнения в наихудшем случае для алгоритма Гиббса — Пула — Стокмейера равно $O(n^3)$, но его практическая производительность близка к линейной.

Программы полного перебора могут точно найти минимальную полосу ленты посредством поиска с возвратом в наборе из всех $n!$ возможных перестановок вершин. Пространство поиска можно значительно разредить с помощью отсечения. Для этого нужно начать с хорошего эвристического решения задачи уменьшения ширины ленты и поочередно добавлять вершины в крайние слева и справа свободные позиции в частичной перестановке.

### Реализации

Код, написанный Дель Корсо (Del Corso) и Манзини (Manzini), для поиска точных решений задач минимизации ширины ленты (см. [CM99]) можно загрузить с веб-сайта **https://people.unipmn.it/manzini/bandmin**. Улучшенные способы решения задачи на основе целочисленного программирования были разработаны Капрарой (Caprara) и Салазар-Гонзалесом (Salazar-Gonzalez), см. [CSG05]. Их реализацию алгоритма метода ветвей и границ на языке C можно загрузить из раздела **Algorithm Repository** веб-сайта этой книги (**www.algorist.com**).

Библиотека NetLib содержит реализации на языке FORTRAN алгоритмов Катхилла — Макки (см. [CM69]) и Гиббса — Пула — Стокмейера (см. [GPS76]). Обзор экспериментальных оценочных тестов этих и других алгоритмов на наборе из 30 матриц рассматривается в публикации [Eve79b]. Согласно этим тестам бессменным лидером является алгоритм Гиббса — Пула — Стокмейера. В докладе [Pet03] предоставляются результаты всестороннего изучения эвристических алгоритмов для решения задачи минимального линейного размещения. Соответствующий код и данные доступны по адресу **http://www.lsi.upc.edu/~jpetit/MinLA/Experiments/**.

#### *Примечания*

Отличный обзор современных алгоритмов решения задачи минимизации ширины ленты и родственных задач на графах представлен в [DPS02]. Обзор решений задачи минимизации ширины ленты для графов и матриц вплоть до 1981 года можно найти в [CCDG82]. Эвристические алгоритмы специального назначения явились предметом всестороннего изучения, что свидетельствует о их важности в численных расчетах. В докладе [Eve79b] приведено не менее полусотни различных алгоритмов минимизации ширины ленты.

Сложность задачи минимизации ширины ленты впервые была установлена в докладе [Pap76b], а ее сложность на деревьях с максимальной степенью 3 — в докладе [GGJK78]. Для задачи минимизации ленты фиксированной ширины $k$ существуют алгоритмы с полиномиальным временем исполнения (см. [Sax80]). Для общей задачи существуют аппроксимирующие алгоритмы с гарантированным полилогарифмическим временем исполнения (см. [BKRV00]). Вне рамок общей задачи временная сложность трудно поддается оценке (см. [DFU11]).

### Родственные задачи

Решение линейных уравнений (см. *разд. 16.1*), топологическая сортировка (см. *разд. 18.2*).

## 16.3. Умножение матриц

**Вход.** Матрица $A$ размером $x \times y$ и матрица $B$ размером $y \times z$.

**Задача.** Вычислить матрицу $A \times B$ размером $x \times z$ (рис. 16.3).

### Обсуждение

Умножение матриц является фундаментальной задачей линейной алгебры. Значение этой задачи для комбинаторных алгоритмов определяется ее эквивалентностью многим другим задачам, включая транзитивное замыкание и сокращение, синтаксический разбор, решение системы линейных уравнений и обращение матриц. Создание быстрого

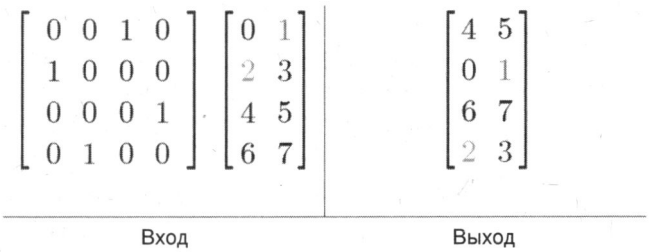



**Рис. 16.3.** Умножение матриц

алгоритма умножения матриц влечет за собой появление быстрых алгоритмов для решения всех этих задач. Умножение матриц возникает как самостоятельная задача при вычислении результатов таких координатных преобразований, как масштабирование, вращение и перемещение в робототехнике и компьютерной графике.

Умножение матриц часто задействуется для решения задач перегруппировки данных, вместо использования для этого жестко запрограммированной логики. Умножение на единичную матрицу ничего не делает. Но посмотрите на матрицы входа и выхода на рис. 16.3. Обратите внимание на то, что в результате умножения на матрицу перестановок (на рис. 16.3, *слева*) строки выходной матрицы были перегруппированы. Такие операции можно выполнять ошеломительно быстро, используя эффективные библиотеки для умножения разреженных матриц.

В листинге 16.1 представлен псевдокод алгоритма, вычисляющего произведение матрицы $A$ размером $x \times y$ и матрицы $B$ размером $y \times z$ за время $O(xyz)$.

**Листинг 16.1. Умножение матриц**

```
Инициализируем массив M[i, j] нулями для всех 1 ≤ i ≤ x и i ≤j ≤ z
   for i = 1 to x do
      for j = 1 to z
         for k = 1 to y
            M[i, j] = M[i,j] + A[i, k] · A[k, j]
```

Реализация этого алгоритма на языке C приводится в листинге 2.4 (см. *разд. 2.5.4*). Казалось бы, этот простой алгоритм на практике превзойти трудно. Но обратите внимание, что три его цикла можно переставить произвольным образом, не повлияв на конечный результат. После такой перестановки изменятся характеристики обращений к памяти и вследствие этого эффективность использования кэша. Можно ожидать, что разброс во времени исполнения между шестью возможными реализациями (перестановками циклов) этого алгоритма достигнет 10–20%, но уверенно предсказать наилучшую из них (обычно *ikj*) нельзя, не выполнив ее на конкретном компьютере с конкретными матрицами.

При умножении двух матриц с шириной ленты, равной $b$, возможно ускорение времени исполнения до $O(xbz)$. Нулевые элементы не участвуют в вычислении произведения матриц, а все ненулевые элементы матриц $A$ и $B$ должны быть расположены среди позиций $b$ элементов главных диагоналей.

Использование рекуррентных методов типа «разделяй и властвуй» позволяет получить еще более быстрые алгоритмы для умножения матриц. Но эти алгоритмы трудно программировать, они требуют матриц очень больших размеров, а также обладают меньшей вычислительной устойчивостью. Наиболее известным из этих алгоритмов является алгоритм Штрассена с временем исполнения $O(n^{2,81})$. Экспериментальные результаты (которые рассматриваются далее) расходятся в определении критической точки, в которой производительность алгоритма Штрассена начинает превышать производительность простого кубического алгоритма, но можно считать, что это происходит при $n \approx 100$.

Однако при умножении цепочки из более чем двух матриц есть лучший способ сэкономить на вычислениях. Вспомните, что при умножении матрицы размером $x \times y$ на матрицу размером $y \times z$ образуется матрица размером $x \times z$. То есть при умножении цепочки матриц слева направо могут образовываться большие промежуточные матрицы, вычисление каждой из которых занимает много времени. Умножение матриц не обладает перестановочным свойством, но имеет ассоциативное, поэтому элементы цепочки можно заключить в скобки любым удобным способом, что не повлияет на конечный результат умножения. Оптимальный порядок скобок можно создать с помощью стандартного алгоритма динамического программирования. Однако такая оптимизация может быть оправданной только в том случае, когда матрицы далеки от квадратных, а их перемножение выполняется достаточно часто. Обратите внимание, что оптимизации подвергаются размеры измерений в цепочке, а не сами матрицы. Если же все матрицы одинакового размера, то такое улучшение невозможно.

Умножение матриц имеет особенно интересную трактовку в задаче подсчета количества путей между двумя вершинами графа. Пусть $A$ представляет матрицу смежности графа $G$ — т. е. $A[i, j] = 1$, если вершины $i$ и $j$ соединены ребром, и $A[i, j] = 0$ в противном случае. Теперь рассмотрим квадрат этой матрицы: $A^2 = A \times A$. Если $A^2[i, j] \geq 1$, то это означает, что должна существовать такая вершина $k$, для которой $A[i, k] = A[k, j]$, поэтому длина пути от вершины $i$ к вершине $k$ и к вершине $j$ равна 2. В более общем смысле, $A^k[i, j]$ подсчитывает количество путей длиной ровно $k$ между вершинами $i$ и $j$. При этом подсчете учитываются непростые пути, т. е. пути с повторяющимися вершинами — например, путь, проходящий через последовательность вершин $i$, $k$, $i$ и $j$.

### Реализации

В работе [DN07] описывается очень эффективный код для умножения матриц, в котором в оптимальной точке выполняется переключение с алгоритма Штрассена на кубический алгоритм. Эту реализацию можно загрузить с веб-сайта **http://www.fastmmw.com**. В соответствующих экспериментах было установлено значение точки пересечения $n \approx 128$, в которой производительность алгоритма Штрассена начинает превосходить производительность кубического алгоритма (см. [BLS91], [CR76]).

Таким образом, кубический ($O(n^3)$) алгоритм будет, вероятнее всего, самым приемлемым выбором для матриц не очень большого размера. Лучшей библиотекой процедур линейной алгебры является LAPACK — более поздняя версия библиотеки LINPACK (см. [DMBS79]), которая содержит ряд процедур для умножения матриц. Эти коды на языке FORTRAN являются частью библиотеки NetLib.

### Примечания

Алгоритм Винограда (Winograd, см. [Win68]) для быстрого умножения матриц уменьшает количество операций умножения наполовину по сравнению с обычным алгоритмом. Это можно считать успешным результатом, несмотря на необходимость дополнительных накладных расходов (см. [DN09]).

По моему мнению, история теоретической разработки алгоритмов берет начало с публикации алгоритма Штрассена (см. [Str69]) для умножения матриц с временем исполнения $O(n^{2,81})$. Впервые улучшение алгоритма в асимптотическом отношении стало целью, заслуживающей самостоятельного исследования. Последующие улучшения алгоритма Штрассена становились все менее практичными. В настоящее время наилучшим алгоритмом для умножения матриц является алгоритм Уильямса (см. [Wil12]) с временной сложностью, составляющей $O(n^{2,3727})$, что в $n^{0,003}$ раз быстрее, чем предыдущий чемпион, алгоритм Копперсмита — Винограда (см. [CW90]). При этом высказываются предположения, что достаточно $\Theta(n^2)$.

Задачу умножения булевых матриц можно свести к общей задаче умножения матриц (см. [CLRS09]). В алгоритме четырех русских (four-Russians algorithm) для умножения булевых матриц (см. [ADKF70]) применяется предварительная обработка с целью создания всех подмножеств lg $n$ строк для ускорения доступа при выполнении собственно операции умножения, что позволяет получить время исполнения $O(n^3/\lg n)$. Дополнительная предварительная обработка может улучшить это время до $O(n^3/\lg^2 \mathrm{n})$ (см. [Ryt85]).

Создание эффективных алгоритмов для умножения матриц требует тщательного управления кэшем. Исследования по этому вопросу см. в [BDN0l] и [HUW02].

Обратной операцией умножения матриц является факторизация матриц — приведение матрицы $M$ к таким матрицам $A$ и $B$, чтобы $M = A \cdot B$. В качестве примера операции факторизации матриц можно назвать LU-разбиение, но в настоящее время проявляется повышенный интерес к низкоразмерной факторизации матриц признаков для использования в науке о данных и машинном обучении (см. [KBV09]).

Помимо подсчета путей, интерес в квадратах графов распространяется и на другие задачи. В работе [Fle74] приводится доказательство, что квадрат любого двусвязного графа содержит гамильтонов цикл. Алгоритмы для нахождения квадратных корней графов — т. е. нахождения матрицы $A$, имея $A^2$, рассматриваются в [LS95].

Хорошее описание алгоритма для умножения цепочки матриц представлено в [BvG99] и [CLRS09] в виде стандартного хрестоматийного примера динамического программирования.

### Родственные задачи

Решение линейных уравнений (см. *разд. 16.1*), поиск кратчайшего пути (см. *разд. 18.4*).

## 16.4. Определители и перманенты

**Вход.** Матрица $M$ размером $n \times n$.

**Задача.** Найти определитель $|M|$ или перманент $perm(M)$ матрицы $M$ (рис. 16.4).

### Обсуждение

Определители матриц — это понятие из линейной алгебры, которое можно использовать для решения многих задач в этой области. В частности, таких:

- проверка, является ли матрица вырожденной — т. е. проверка существования матрицы, обратной имеющейся. Матрица $M$ является вырожденной, если $|M| = 0$;
- проверка, располагается ли множество из $d$ точек на плоскости в менее чем $d$ измерениях. Если да, то определяемая ими система уравнений является вырожденной, поэтому $|M| = 0$;
- проверка, находится ли точка справа или слева от линии или плоскости. Эта задача сводится к проверке знака определителя (см. *разд. 20.1*);
- вычисление площади или объема треугольника, четырехгранника или другого многогранника. Эти величины являются функцией значения определителя (см. *разд. 20.1*).

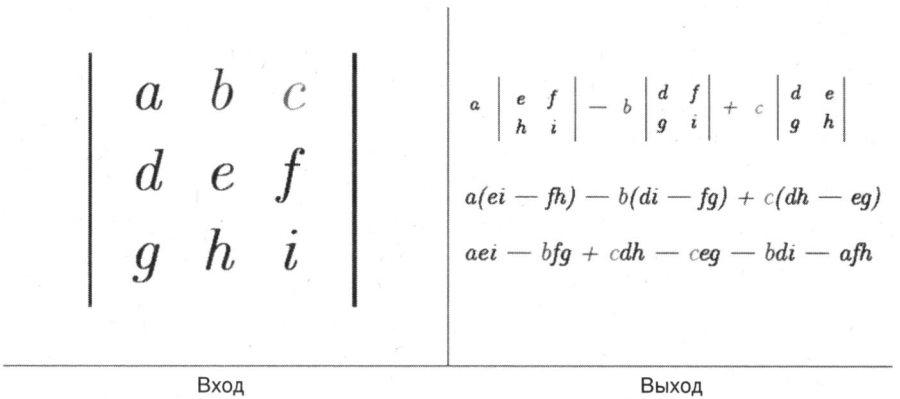



**Рис. 16.4.** Вычисление определителя матрицы

Определитель матрицы $M$ — это сумма всех $n!$ возможных перестановок $\pi_i$ из $n$ столбцов матрицы:

$$|M| = \sum_{i=1}^{n!} (-1)^{sign(\pi_i)} \prod_{j=1}^{n} M[j, \pi_j],$$

где $sign(\pi_i)$ — количество пар элементов, расположенных не по порядку (называемых *инверсиями*) в перестановке $\pi_i$.

Прямая реализация этого определения дает алгоритм с временем исполнения $O(n!)$ точно так же, как и метод алгебраических дополнений. Алгоритмы с лучшей производительностью для вычисления определителей основаны на LU-разложении, рассматриваемом в *разд. 16.1*. При этом подходе определитель матрицы $M$ вычисляется просто как произведение диагональных элементов LU-разложения матрицы, для чего требуется время $O(n^3)$.

В комбинаторных задачах часто встречается похожая на определитель функция, называемая *перманентом*. Например, перманент матрицы смежности графа подсчитывает количество совершенных паросочетаний графа. Перманент матрицы $M$ вычисляется следующим образом:

$$perm(M) = \sum_{i=1}^{n!} \prod_{j=1}^{n} M[j, \pi_j].$$

Единственное отличие перманента от определителя — то, что все произведения положительные.

Удивительно — задача вычисления перманента является NP-полной, хотя определитель можно с легкостью вычислить за время $O(n^3)$. Фундаментальная разница между этими двумя функциями состоит в том, что $det(AB) = det(A) \times det(B)$, в то время как $perm(AB) \neq perm(A) \times perm(B)$. Существуют алгоритмы вычисления перманента за время $O(n^2 2^n)$, что оказывается значительно быстрее, чем время $O(n!)$. То есть вычисление перманента матрицы размером 20×20 является вполне реальной задачей.

### Реализации

Пакет инструментов для линейной алгебры LINPACK содержит множество процедур на языке FORTRAN для вычисления определителей. Библиотека для научных расчетов JScience содержит обширный пакет инструментов для линейной алгебры (включая определители). Еще одним пакетом для работы с матрицами на языке Java является библиотека JAMA. Ссылки на эти и многие другие библиотеки можно найти на сайте **http://math.nist.gov/javanumerics**.

Эффективная процедура на языке FORTRAN для вычисления перманентов матриц приводится в книге [NW78]. Подробности см. в *разд. 22.1.9*. В работе [Cas95] описывается процедура на языке C для вычисления перманентов на основе подсчета структур Кекуле (Kekule structures) в вычислительной химии.

Две разные процедуры для аппроксимации перманента разработаны профессором математики Университета Мичигана Александром Барвинком. Первый, на основе работы [BS07], предоставляет коды для аппроксимации перманента и гафниана матрицы, а также количества остовных деревьев графа. Подробности — на веб-сайте **http://www.math.lsa.umich.edu/~barvinok/manual.html**. Второй, на основе работы [SB01], может выдавать приблизительное значение перманента матрицы размером 200×200 за секунды. Подробности — по адресу **http://www.math.lsa.umich.edu/~barvinok/code.html**.

#### *Примечания*

Правило Крамера сводит задачи обращения матриц и решения систем линейных уравнений к задаче вычисления определителей. Но алгоритмы на основе LU-разложения работают быстрее. Описание правила Крамера см. в [BM53].

Определители можно вычислять за время $O(n^3)$, используя метод быстрого умножения матриц, как показано в книге [AHU83]. Такие алгоритмы обсуждаются в *разд. 16.3*. Разработчиком быстрого алгоритма для вычисления знака определителя является Кларксон (Clarkson); см. [Cla92].

В работе [Val79] было доказано, что задача вычисления перманента является #P-полной, где #P означает класс задач, решаемых на «счетной» машине за полиномиальное время. «Счетная» машина возвращает количество разных решений задачи. Подсчет количества гамильтоновых циклов в графе является #P-полной задачей и, очевидно, NP-сложной (а, возможно, еще сложнее), поскольку ненулевой результат подсчета означает, что граф является гамильтоновым. Но задачи подсчета могут быть #P-полными, даже если соответствующую задачу разрешимости можно решить за полиномиальное время с использованием перманента и совершенных паросочетаний.

Основным справочником по перманентам является [Min78]. В книге [NW78] представлена разновидность алгоритма для вычисления перманента за время $O(n^2 2^n)$.

За последнее время были разработаны вероятностные алгоритмы для приблизительного вычисления перманента, вершиной развития которых является полиномиальный рандомизированный алгоритм, обеспечивающий сколь угодно точную аппроксимацию за время, полиномиально зависящее от размера матрицы входа и допускаемого уровня погрешности (см. [JSV04]).

### Родственные задачи

Решение линейных уравнений (см. *разд. 16.1*), паросочетание (см. *разд. 18.6*), геометрические примитивы (см. *разд. 20.1*).

## 16.5. Условная и безусловная оптимизация

**Вход.** Функция $f(x_1, ..., x_n)$.

**Задача.** Найти точку $p = (p_1, ..., p_n)$, в которой функция $f$ достигает минимума или максимума (рис. 16.5).

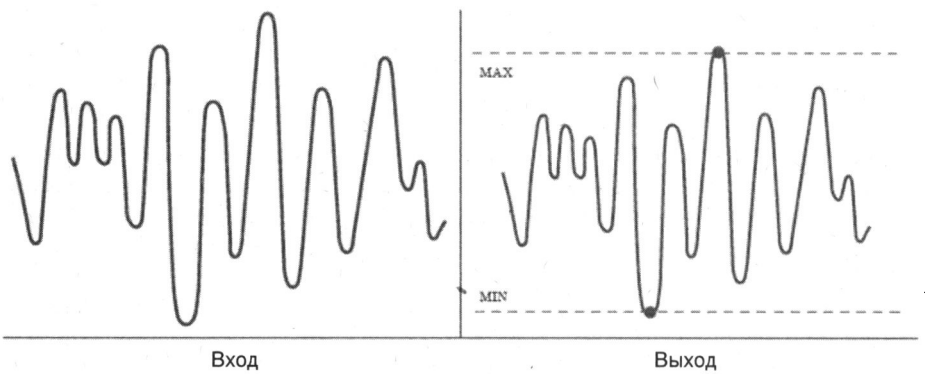



**Рис. 16.5.** Поиск минимума и максимума функции

### Обсуждение

Из 75 задач этого каталога важность рассматриваемой здесь задачи со времени предыдущего издания книги возросла наиболее значительно. Алгоритмы для решения задач выпуклой и невыпуклой оптимизации входят в число алгоритмов, напрямую связанных с машинным обучением — от линейной регрессии до глубокого обучения.

Задача оптимизации возникает, когда целевую функцию требуется настроить на оптимальную производительность. Допустим, что мы разрабатываем программу для инвестиций на бирже. При этом у нас есть некоторые финансовые данные, которые мы можем анализировать: отношение цены акции к чистой прибыли, процентную ставку и цену акции. Все эти величины являются функцией времени $t$. Самое сложное здесь — определение коэффициентов следующей формулы:

$$\text{качество_акций}\,(t) =$$
$$= c_1 \times \text{цена}\,(t) + c_2 \times \text{процентная_ставка}\,(t) + c_3 \times \text{отношение_цены_к_прибыли}\,(t).$$

Нам нужно найти такие значения $c_1$, $c_2$ и $c_3$, которые оптимизируют значение функции качества акций на основе исторических данных. Подобные вопросы возникают при настройке оценочных функций для любой задачи распознавания закономерностей или машинного обучения.

Задачи безусловной оптимизации также возникают и в научных расчетах. Физические системы, от протеиновых цепочек до галактик, естественно стремятся минимизировать свою «энергию», или «потенциальную функцию». Поэтому в программах, которые эмулируют природу, потенциальная функция часто определяется путем присваивания некоторого значения каждой возможной конфигурации объектов с последующим выбором из всех этих конфигураций той, у которой потенциал минимален.

Глобальные задачи оптимизации, как правило, сложны, и для их решения существует множество подходов. Чтобы выбрать правильный путь решения таких задач, ответьте на следующие вопросы.

- *Задача какого типа решается (условной или безусловной оптимизации)?*

  В задачах безусловной оптимизации нет никаких ограничений на значения параметров — кроме условия, чтобы они максимизировали значение функции. Но во многих приложениях требуется накладывать ограничения на эти параметры, вследствие чего некоторые значения, которые в противном случае могли бы дать оптимальное решение, становятся запрещенными. Например, предприятие не может иметь в штате меньше чем нулевое количество сотрудников, невзирая на то, сколько денег они могли бы сэкономить, имея, скажем, 100 сотрудников. Задачи условной оптимизации обычно требуют подхода, основанного на математическом программировании наподобие линейного программирования, рассматриваемого в *разд. 16.6*.

- *Можно ли оптимизируемую функцию выразить формулой?*

  Иногда требуется вычислить максимум или минимум функции, представленной в виде алгебраической формулы, — например, найти минимальное значение:

  $$f(n) = n^2 - 6n + 2^{n+1}.$$

  В таком случае нужно взять производную этой функции и вычислить, для каких точек $p'$ производная будет равна нулю, т. е. $f'(p') = 0$. Ими окажутся точки локального максимума или минимума, различить которые можно, взяв вторую производную или просто подставив значение $p'$ в функцию $f$ и оценив полученный результат. Следует иметь в виду, что ситуация усложняется в случае с многомерными функциями. Такие системы, как Mathematica и Maple, весьма успешно вычисляют производные, хотя эффективное использование компьютерных приложений вычислительной алгебры в определенной степени сродни шаманству. Тем не менее имеет смысл попробовать эти системы, поскольку с их помощью можно как минимум создать диаграмму оптимизируемой функции, чтобы получить наглядное представление о ее поведении.

- *Является ли функция выпуклой?*

  Основная опасность, подстерегающая нас при глобальной оптимизации, — это ловушка локального оптимума. Рассмотрим задачу поиска наивысшей точки горного массива. Если этот массив состоит всего лишь из одной горы правильной формы, то наивысшую точку можно найти, просто идя вверх по этой горе. Но если мы имеем

дело с горным хребтом, содержащим несколько высоких гор, то найти наивысшую возможную точку будет труднее.

*Выпуклые функции* обладают ровно одним максимумом (или минимумом), что в реальном мире соответствуют одной горной вершине. Самый быстрый путь на вершину горы (или в самую нижнюю точку долины) определяется поисковым алгоритмом градиентного спуска, анализирующим частичные производные (наклон). Оптимум достигается в точке, в которой значения производных равны нулю. Алгоритмы выпуклых оптимизаций можно использовать для быстрого решения массивных задач — даже в пространствах с большим количеством размерностей. Возьмем для примера задачу вычисления множества коэффициентов, минимизирующих ошибку при подборе эмпирической кривой для задачи линейной регрессии при, скажем, 1000 входных переменных, или размерностей. В этом случае оптимальное решение (минимальная точка) определяет 1000 параметров. Если целевая функция выпуклая (как в случае с линейной регрессией), задача быстро решается при помощи поискового алгоритма градиентного спуска.

Как узнать, является ли та или иная функция выпуклой? Для этого нужно выполнить анализ ее производных, но эта тема выходит за рамки моей книги. Однако если кто-то умный говорит вам, что функция выпуклая, то следует поверить ему на слово.

- *Насколько непрерывной или плавной является функция?*

Даже если функции и не выпуклая, шансы хорошие, что она будет, по крайней мере, плавной. *Плавность графика функции* определяется свойством, согласно которому значения точек, расположенных в тесном соседстве с точкой $p$, должны быть близкими к значению этой точки. Плавность функции подразумевается при любом выполнении процедуры поиска. Если бы высота в любой точке кривой определялась бы полностью произвольным значением, то найти оптимальную высоту было бы невозможно никаким алгоритмом, кроме как проверкой каждой точки без исключения.

- *Насколько затратно вычисление функции в определенной точке?*

Иногда вместо формулы для вычисления значения $f$ в какой-либо точке $X$ можно использовать программу или процедуру. Тогда мы имеем возможность, вызвав эту процедуру, рассчитать по требованию значение любой точки, в результате чего можем попробовать определить оптимальное значение, вычислив значения достаточно большого количестве точек в представляющем интерес районе.

Но наша свобода поиска оптимального значения зависит от того, насколько эффективно мы можем вычислять значение $f(X)$. При дорогостоящей процедуре вычисления значения точки наилучшим подходом будет простой поиск по сетке параметров. Выполняется такой поиск следующим образом. Предположим, что мы можем позволить себе протестировать $m \approx s^k$ возможных точек, где $s$ — некое целое число. Идентифицируем наименьшее и наибольшее возможные значения вдоль каждой из $k$ размерностей, а затем разбиваем каждый диапазон на $s$ равномерно распределенных значений. Существует $s^k$ отдельных точек, которые можно определить, выбирая одно значение из каждого $k$ диапазонов, — такие точки грубо покрывают множество возможностей. Вычисляем значение $f(X)$ в каждой из этих точек и выбираем наи-

лучшую из них, приняв ее за «оптимальную». Полученную таким образом точку можно использовать в качестве начальной точки для более систематического поиска.

Такая ситуация возникает при настройке оценочных функций для игр. Допустим, что у нас имеется функция $f(x_1, ..., x_n)$, оценивающая ситуацию на шахматной доске, где $x_1$ — стоимость пешки, $x_2$ — стоимость слона и т. д. Чтобы выбрать оптимальные значения набора коэффициентов для этой функции, нужно сыграть с ней большое количество игр или протестировать ее на библиотеке известных игр. Очевидно, что такой подход отнимает много времени, поэтому надо стремиться минимизировать количество вычислений функции, выполняемых при оптимизации коэффициентов.

В наиболее эффективных алгоритмах выпуклой оптимизации для поиска локального оптимума применяются производные и частные производные, позволяющие выяснить направление, в котором следует двигаться из какой-либо точки, чтобы как можно быстрее достичь максимального или минимального значения функции. Иногда такие производные можно рассчитать аналитическим способом, или же значение производной может быть вычислено приблизительно — путем определения разности между значениями функции в соседних точках. Для поиска локального оптимума было разработано много разнообразных методов *быстрейшего спуска* (steepest descent) и *сопряженных градиентов* (conjugate gradient), которые во многом подобны числовым алгоритмам поиска корня.

В задаче условной оптимизации часто трудно найти точку, удовлетворяющую всем ограничениям. Один из возможных подходов к решению — использование метода безусловной оптимизации с добавлением штрафных санкций в зависимости от количества нарушенных ограничивающих условий. В этом и заключается суть *лагранжевой релаксации*, если вы еще помните свой курс высшей математики. Определение правильной функции начисления штрафных санкций зависит от конкретной задачи, но часто имеет смысл менять штрафные санкции в ходе процесса оптимизации. В конечном счете штрафные санкции должны быть значительными, чтобы обеспечить удовлетворение всех ограничивающих условий.

Метод имитации отжига является достаточно простым и устойчивым к ошибкам подходом к решению задачи условной оптимизации — особенно когда оптимизация выполняется по комбинаторным структурам (перестановкам, графам, подмножествам и т. п.). Технология метода имитации отжига описывается в *разд. 12.6.3*.

Стоит попробовать несколько разных методов для задачи оптимизации. Прежде чем пытаться создать свой собственный метод, поэкспериментируйте с описанными далее реализациями. Четкие описания таких алгоритмов можно найти во многих книгах по числовым алгоритмам — в частности, в книге [PFTV07].

### Реализации

Предмет безусловной и условной оптимизации является достаточно сложным, вследствие чего было издано несколько руководств в помощь разработчикам. Лучшим из этих руководств является «Decision Tree for Optimization Software», доступное по адресу **http://plato.asu.edu/guide.html**.

Система NEOS (Network-Enabled Optimization System — оптимизирующая система с поддержкой работы в сетевой среде) предоставляет уникальный сервис — возможность решать задачи дистанционно на компьютерах и программном обеспечении Висконсинского института открытий (Wisconsin Institute of Discovery). Поддерживается как линейное программирование, так и безусловная оптимизация. Подробности вы найдете по адресу **https://neos-server.org**.

Также доступны реализации общего назначения метода имитации отжига, которые, скорее всего, явятся наилучшей отправной точкой для экспериментов с этой технологией условной оптимизации. Можете свободно пользоваться моим кодом этого метода (см. *разд. 12.6.3*). Особенно популярной реализацией метода имитации отжига является реализация Adaptive Simulated Annealing на языке C, которую можно загрузить с вебсайта **http://asa-caltech.sourceforge.net/**.

#### *Примечания*

В работах [Ber15], [BV04] и [Nes13] предоставляется исчерпывающее рассмотрение выпуклой оптимизации, включая методы на основе алгоритма градиентного спуска. Сама область безусловной оптимизации и лагранжевой релаксации также является предметом многих книг, включая [Ber82] и [PFTV07].

Полные целевые функции, используемые для решения задач машинного обучения, часто имеют линейный размер обучающих данных, что значительно повышает стоимость вычисления частных производных для градиентного спуска. Намного лучшим подходом будет прикидочно оценить производные для текущей позиции, используя небольшую произвольную выборку обучающих данных. Такие стохастические алгоритмы градиентного спуска рассматриваются в [Bot12]. В число хороших книг по машинному обучению входят так же [Bis06] и [FHT01].

Метод имитации отжига был разработан в качестве современной версии алгоритма Метрополиса (см. [MRRT53]). В обоих алгоритмах используется метод Монте-Карло для вычисления минимального энергетического уровня системы. Хорошее описание всех разновидностей локального поиска, включая метод имитации отжига, представлено в [AL97].

Генетические алгоритмы были разработаны и популяризированы Холландом (Holland) — см. [Hol75] и [Hol92]. Более благожелательные описания генетических алгоритмов, чем приведено в этой книге, можно найти в работах [LP02] и [MF00].

### Родственные задачи

Линейное программирование (см. *разд. 16.6*), выполнимость (см. *разд. 17.10*).

## 16.6. Линейное программирование

**Вход.** Набор $S$ из $n$ линейных неравенств с $m$ переменными:

$$S_i = \sum_{j=1}^{m} c_{ij} x_j \geq b_i, \ 1 \leq i \leq n,$$

и функция линейной оптимизации: $f(X) = \sum_{j=1}^{m} c_j \cdot x_j$.

**Задача.** Найти такой набор переменных $X'$, который максимизирует целевую функцию $f$, при этом выполняя все неравенства $S$ (рис. 16.6).

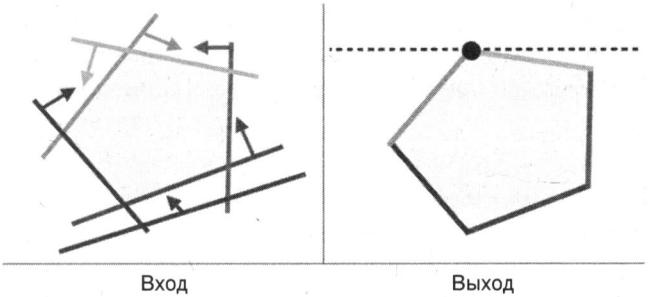



**Рис. 16.6.** Графическое представление задачи линейного программирования

### Обсуждение

Задачи линейного программирования представляют собой важнейший тип задач из области математической оптимизации и исследования операций.

Линейное программирование имеет следующие приложения:

- *распределение ресурсов.*

  В этой задаче требуется вложить определенную сумму, чтобы максимизировать полученную прибыль. Часто возможные опции капиталовложения, выплаты и расходы можно выразить в виде системы линейных неравенств таким образом, чтобы максимизировать потенциальный доход при указанных ограничивающих условиях. Крупномасштабные задачи линейного программирования постоянно возникают в авиакомпаниях и других корпорациях;

- *аппроксимация решения несовместных систем уравнений.*

  Система $n$ линейных уравнений с $m$ неизвестными $x_i$, $1 \le i \le m$, называется *переопределенной* (overdetermined), если $n > m$. Переопределенные системы часто являются *несовместными* (inconsistent) — т. е. не существует набора значений, которые одновременно удовлетворяют всем уравнениям. Чтобы найти набор значений, наиболее подходящий для решения уравнений, мы можем заменить каждую переменную $x_i$ выражением $x'_i + \varepsilon_i$ и решить новую систему, минимизируя сумму элементов вектора ошибок;

- *алгоритмы на графах.*

  Многие из рассматриваемых в этой книге задач на графах, включая задачи поиска кратчайшего пути, паросочетания в двудольных графах и потоков в сети, можно решить как частные случаи задачи линейного программирования. Большинство других задач, включая задачу коммивояжера, покрытия множества и задачу о рюкзаке, можно решить посредством *целочисленного линейного программирования.*

Стандартным алгоритмом линейного программирования является *симплекс-метод.* При его использовании каждое ограничивающее условие задачи отсекает часть пространства возможных решений. Наша цель — найти такую точку в оставшемся после отсечения пространстве решений, которое максимизирует (или минимизирует) функцию $f(X)$. Вращая должным образом пространство решений, эту оптимальную точку можно всегда сделать наивысшей точкой области. Область (симплекс), определенная

пересечениями линейных ограничений, является выпуклой, поэтому, если только мы уже не находимся на самом верху, всегда существует соседняя вершина, расположенная выше. Если мы не можем найти более высокую соседнюю точку, значит, мы нашли оптимальное решение.

В то время как базовый симплексный алгоритм не особенно сложен, создание его эффективной реализации, способной решать задачи линейного программирования на больших входных экземплярах, требует значительного мастерства. Входные экземпляры значительных размеров, как правило, обладают *разреженностью* (т. е. во многих неравенствах имеется малое количество переменных), что влечет за собой необходимость использования сложных структур данных. Необходимо помнить о важности вопросов вычислительной устойчивости и надежности, а также выбора наилучшей следующей точки для посещения (так называемые *правила выбора переменных*, pivot rules). Впрочем, существуют сложные алгоритмы *методов внутренней точки* (interior-point methods), в которых переход к следующей точке осуществляется через внутреннюю область симплекса, а не по поверхности. Во многих приложениях эти алгоритмы превосходят алгоритмы симплекс-метода.

Подводя итоги, можно сказать о линейном программировании следующее: намного выгоднее использовать существующие реализации инструментов линейного программирования, нежели пытаться создать свою собственную программу. Кроме того, надежнее приобрести платное решение, чем искать бесплатное в Интернете. Задача линейного программирования настолько важна в экономических приложениях, что коммерческие реализации превосходят бесплатные во всех отношениях.

Приведем вопросы, возникающие при решении задач линейного программирования.

- *Ограничены ли какие-либо переменные целочисленными значениями?*

  Даже если согласно вашей модели максимальную прибыль можно получить, ежедневно отправляя из Нью-Йорка в Вашингтон 6,54 рейса, реализовать это на практике невозможно. Такие переменные, как, например, количество авиарейсов, часто обладают естественными целочисленными ограничениями. Задача линейного программирования называется *целочисленной*, если все ее переменные обладают целочисленными ограничениями, и *смешанной*, если такие ограничения наложены не на все переменные, а только на некоторые из них.

  К сожалению, поиск оптимального решения целочисленной или смешанной задачи является NP-полной задачей. Однако существуют методы целочисленного программирования, которые на практике работают достаточно хорошо. Например, *методы секущей плоскости* сначала решают задачу как линейную, после чего добавляют дополнительные ограничивающие условия, чтобы наложить требование целочисленности в окрестности оптимального решения, после чего решают задачу повторно. В результате достаточного количества итераций оптимальная точка получившейся задачи линейного программирования совпадает с оптимальной точкой первоначальной целочисленной задачи. Как и в большинстве экспоненциальных алгоритмов, время исполнения алгоритмов целочисленного программирования зависит от сложности конкретного экземпляра задачи, поэтому предсказать его невозможно.

- *Чего в задаче больше — переменных или ограничивающих условий?*

  Любую задачу линейного программирования с $m$ переменными и $n$ неравенствами можно записать в виде эквивалентной *двойственной* задачи с $n$ переменными и $m$ неравенствами. Важно знать это обстоятельство, поскольку время исполнения обоих указанных вариантов процедуры решения может значительно различаться. Как бы там ни было, задачи линейного программирования, в которых переменных больше, чем ограничивающих условий, следует решать прямым способом. Если же количество ограничивающих условий превышает количество переменных, то обычно лучше решать двойственную задачу линейного программирования или (что равносильно) применять двойственный симплекс-метод к первоначальной задаче.

- *Как поступить, если функция оптимизации или ограничивающие условия не являются линейными?*

  Задача аппроксимации кривой методом наименьших квадратов заключается в поиске прямой, для которой сумма квадратов расстояний между каждой заданной точкой и ею минимальна. При постановке этой задачи естественная целевая функция является квадратичной, а не линейной. Хотя для аппроксимации кривой методом наименьших квадратов существуют быстрые алгоритмы, общая задача *квадратичного программирования* является NP-полной.

  Существуют три возможных подхода к решению задачи нелинейного программирования. Самый лучший — попытаться смоделировать ее (если это возможно) каким-либо другим способом, как в задаче аппроксимации кривой методом наименьших квадратов. Также можно попытаться найти специальные реализации для квадратичного программирования. Наконец, можно смоделировать задачу в виде задачи условной или безусловной оптимизации и попытаться решить ее с помощью реализаций, рассматриваемых в *разд. 16.5*.

- *Что делать, если моя модель не совпадает с форматом входа имеющейся у меня реализации решения задачи линейного программирования?*

  Многие реализации решений задач линейного программирования принимают модели только в так называемом *стандартном формате*, где все переменные должны быть неотрицательными, целевая функция — минимизирована, а все ограничивающие условия — сформулированы в виде равенств (а не неравенств).

  В этом требовании нет ничего страшного. Известны преобразования для приведения произвольных задач линейного программирования к стандартному формату. В частности, чтобы преобразовать задачу максимизации в задачу минимизации, нужно просто поменять знак коэффициента целевой функции на противоположный. Остальные несоответствия можно исправить, добавляя в модель *фиктивные переменные* (slack variable). Необходимые подробности вы найдете в любом учебнике по линейному программированию. Кроме того, вопрос можно решить посредством добавления хорошего интерфейса к программной реализации, воспользовавшись каким-либо языком моделирования, — например, языком AMPL.

### Реализации

Существуют, по крайней мере, три хорошие бесплатные реализации для решения задач линейного программирования. Приложение lp_solve, написанное Майклом Беркелаа-

ром (Michel Berkelaar) на языке ANSI C, работает как с целочисленными, так и со смешанными задачами. Загрузить приложение можно с веб-сайта **http://lpsolve.sourceforge.net/5.5/**. Существует также огромное сообщество пользователей этого приложения. Другая библиотека, CLP, для решения задач линейного программирования симплекс-методом предоставляется проектом Computational Infrastructure for Operations Research и доступна по адресу **http://www.coin-or.org/**. Наконец, пакет GLPK (GNU Linear Programming Kit) предназначен для решения крупномасштабных задач линейного программирования, целочисленного программирования и других родственных задач. Этот пакет можно загрузить с веб-сайта **http://www.gnu.org/software/glpk/**.

Согласно двум сравнительным исследованиям ([GAD+13] и [MT12]) эффективность и качество работы коммерческих пакетов для решения задач линейного программирования намного выше, чем средств с открытым кодом. Но они не могут согласиться друг с другом, какой из рассмотренных пакетов лучше. Подробности вы найдете в указанных докладах.

Система NEOS предоставляет возможность решать задачи дистанционно на компьютерах и программном обеспечении Висконсинского института открытий (Wisconsin Institute of Discovery). Поддерживается как линейное программирование, так и безусловная оптимизация. На эту систему стоит обратить внимание, если вам нужна не программа для решения задачи, а только ответ на задачу. Дополнительную информацию см. на веб-сайте **https://neos-server.org**.

### *Примечания*

Потребность в оптимизации посредством линейного программирования возникла при решении задач материально-технического снабжения во время Второй мировой войны. Симплексный алгоритм был изобретен Джорджем Данцигом (George Danzig) в 1947 году (см. [Dan63]). А Кли (Klee) и Минти (Minty) доказали, что симплексный алгоритм имеет экспоненциальное время исполнения в наихудшем случае, но является очень эффективным на практике (см. [KM72]).

При сглаживающем анализе измеряется сложность алгоритмов в предположении, что их вход содержит небольшой объем помех. Аккуратно построенные экземпляры наихудших случаев для многих задач не выдерживают такой проверки. Эффективность симплекс-метода в практическом применении была объяснена с помощью сглаживающего анализа Даниелем Шпильманом (Daniel Spielman) и Шанг-Хуа Тенгом (Shang-Hua Teng) (см. [ST04]). А Джонатан А. Кельнер (Jonathan A. Kelner) совместно с Даниелем Шпильманом разработали рандомизированный симплексный алгоритм с полиномиальным временем исполнения (см. [KS05b]).

Полиномиальность задач линейного программирования впервые была доказана в 1979 году посредством эллиптического алгоритма (см. [Kha79]). Алгоритм Кармаркара (Karmarkar) на основе метода внутренней точки (см. [Kar84]) является как теоретическим, так и практическим улучшением эллиптического алгоритма, а также соперником симплексного метода. Хорошие описания симплексного и эллиптического алгоритмов приведены в [Chv83], [Gas03] и [MG07].

Полуопределенное программирование применяется для решения задач оптимизации с переменными симметрических положительных полуопределенных матриц, линейной функцией стоимости и линейными ограничивающими условиями. Важные частные случаи включают в себя задачи линейного программирования и задачи выпуклого квадратичного программирования с выпуклыми квадратичными ограничивающими условиями. Полу-

определенное программирование и его использование для решения комбинаторных задач оптимизации обсуждается в публикациях [Goe97] и [VB96].

Задачи линейного программирования являются P-полными с логарифмической сложностью по памяти (см. [DLR79]). Вследствие этого создать параллельный алгоритм для решения задач класса NC, скорее всего, невозможно. (Задача принадлежит классу NC тогда и только тогда, когда ее можно решить на машине PRAM за полилогарифмическое время, используя полиномиальное количество процессоров.) Любая P-полная задача по сведению с логарифмической сложностью по памяти не может быть членом класса NC, если только не выполняется условие P = NC. Всестороннее рассмотрение теории P-полноты, включающее в себя обширный список P-полных задач, представлено в книге [GHR95].

### Родственные задачи

Условная и безусловная оптимизация (см. *разд. 16.5*), потоки в сети (см. *разд. 18.9*).

## 16.7. Генерирование случайных чисел

**Вход.** Либо ничего, либо начальное число (seed).

**Задача.** Сгенерировать последовательность случайных целых чисел (рис. 16.7).

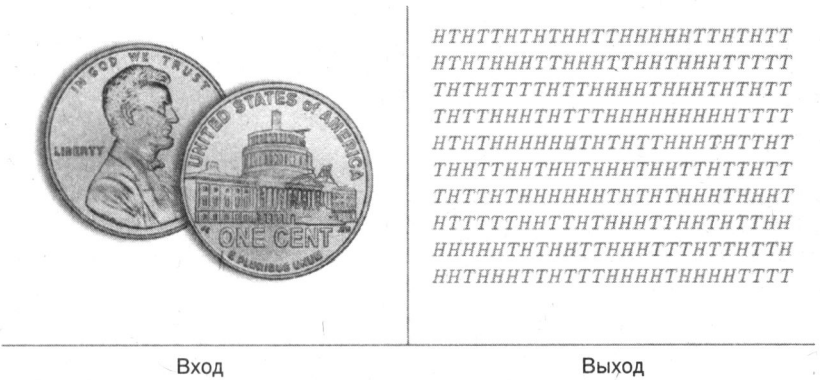



**Рис. 16.7.** Последовательность случайных символов

### Обсуждение

Случайным числам находится место в самых разных интересных и важных приложениях. Они лежат в основе метода имитации отжига и родственных эвристических методов оптимизации. Эмуляция дискретных событий выполняется на потоках случайных чисел и используется для моделирования широкого диапазона задач — от транспортных систем до игры в покер. Случайным образом обычно генерируются различные пароли и ключи шифрования. Рандомизированные алгоритмы для решения задач на графах и геометрических задач коренным образом изменяют эти области и возводят рандомизацию в ранг одного из основополагающих принципов теории вычислительных систем.

К сожалению, задача генерирования случайных чисел выглядит гораздо более легкой, чем она в действительности является. Более того, создать истинно случайное число на

любом детерминистском устройстве, по сути, невозможно. Лучше всего это выразил фон Ньюман (см. [Neu63]): «Любой, кто считает возможным применение арифметических методов для генерирования случайных чисел, несомненно, грешит». Самое лучшее, что мы можем надеяться получить с помощью арифметических методов, — это *псевдослучайные* числа, т. е. последовательность чисел, которые лишь выглядят случайными.

Отсюда вытекают серьезные последствия использования некачественного генератора случайных чисел. В одном получившем известность случае схема шифрования веб-браузера была взломана, поскольку в нем использовалось недостаточно много случайных битов (см. [GW96]). Вследствие использования некачественной функции генерирования случайных чисел результаты моделирования различных ситуаций постоянно получаются неточными или даже полностью неверными. Это такая область, в которой не следует заниматься самодеятельностью, но многие обычно переоценивают свои способности.

Приведем вопросы, касающиеся работы со случайными числами:

- *Можно ли создавать один и тот же набор «случайных» чисел при каждом исполнении программы?*

  Игра в покер, в которой вам каждый раз сдаются одни и те же карты, быстро надоест. Одним из распространенных решений этой проблемы является использование младших битов показаний часов компьютера в качестве начального числа (seed) потока случайных чисел, чтобы при каждом исполнении программы генерировалась другая последовательность.

  Такие методы удовлетворительны для игр, но не для моделирования серьезных ситуаций. Всякий раз, когда вызовы функции генерирования случайных чисел выполняются в цикле, существует вероятность периодичности распределения получаемых случайных чисел. С другой стороны, когда программа выдает разные результаты при каждом ее исполнении, ее отладка серьезно усложняется. В случае сбоя программы вы не сможете отследить ее исполнение, чтобы выяснить причину сбоя. Возможным компромиссом является использование детерминистского генератора псевдослучайных чисел с сохранением между исполнениями программы текущего начального числа в файле. При отладке в этот файл можно будет записывать фиксированное начальное число.

- *Насколько надежен встроенный генератор случайных чисел моего компилятора?*

  Если вам требуются равномерно распределенные случайные числа и требования к точности моделирования не являются критическими, то я рекомендую просто использовать генератор, который встроен в компилятор. Здесь очень легко совершить ошибку, неудачно выбрав начальное число, поэтому вы должны внимательно прочитать руководство пользователя.

  Но если точность результатов моделирования для вас *критична*, то лучше использовать собственный генератор случайных чисел. Однако следует иметь в виду, что определить «на глазок», действительно ли генерируемые последовательности являются случайными, трудно, потому что у многих исследователей имеется искаженное представление о поведении генераторов случайных чисел, и они часто видят за-

кономерности, которых в действительности не существует. Качество генератора случайных чисел следует проверить на нескольких разных тестах и установить статистическую достоверность результатов. Для оценки качества генераторов случайных чисел Национальный институт стандартов и технологий США разработал специальный набор тестов, который рассматривается далее — в *подразделе «Реализации»*.

- *Как реализовать собственный генератор случайных чисел?*

  Стандартный вариант реализации генератора случайных чисел — *линейный конгруэнтный генератор* (linear congruential generator). Это простой и быстрый генератор, который дает достаточно удовлетворительные псевдослучайные числа (при условии задания правильных значений констант). Случайное число $R_n$ является функцией $(n-1)$-го сгенерированного случайного числа:

$$R_n = (aR_{n-1} + c) \bmod m.$$

  Поведение линейного конгруэнтного генератора аналогично поведению рулетки. Длинный путь шарика вдоль окружности колеса, определяемый выражением

$$aR_{n-1} + c,$$

  заканчивается в одной из немногочисленных лунок. Этот «выбор» весьма чувствителен к длине пути (усеченной с помощью выражения mod $m$).

  Для правильного выбора значений констант $a$, $c$, $m$ и $R_0$ была разработана специальная теория. Длина периода в значительной степени является функцией от $m$, значение которого обычно ограничено длиной слова конкретного компьютера.

  Обратите внимание на то, что последовательность чисел, выдаваемая линейным конгруэнтным генератором, начинает повторяться, как только повторяется первое число. Кроме этого, современные компьютеры обладают достаточно высокой скоростью, позволяющей им вызывать функцию генератора $2^{32}$ раз за несколько минут. Таким образом, для любого линейного конгруэнтного генератора на компьютере с длиной слова в 32 бита существует опасность циклического повторения значений, что диктует необходимость в генераторах, имеющих значительно более длинные периоды.

- *Что делать, если нет надобности в таких больших случайных числах?*

  Линейный конгруэнтный генератор $R_n$ создает равномерно распределенную последовательность целых чисел в диапазоне от 0 до $m-1$, которую можно с легкостью масштабировать для генерирования других равномерных распределений. Для генерирования действительных чисел между 0 и 1 служит формула $R_n/m$. Обратите внимание, что получить число 1 посредством этой формулы нельзя, хотя число 0 — можно. Если нужно получить равномерно распределенные целые числа между $l$ и $h$, используется формула

$$\lfloor l + (h - l + 1) R_n / m \rfloor.$$

- *Что делать, если распределение генерируемых случайных чисел должно быть неравномерным?*

  Генерирование случайных чисел согласно той или иной функции неравномерного распределения может быть нелегкой задачей. Самым надежным будет *метод вы-*

*борки с отклонениями* (acceptance-rejection method). Геометрическая область, из которой требуется делать выборку, заключается в ограничивающее окно, а потом в нем выбирается произвольная точка $p$. Это точку в ограничивающем окне можно получить, независимо генерируя произвольные значения ее координат $x$ и $y$. Если полученная точка $p$ лежит в интересующей нас области, то мы принимаем ее как случайно выбранную. В противном случае мы отказываемся от этой точки, и процесс повторяется. Можно сказать, что мы бросаем дротики с закрытыми глазами и засчитываем только те, которые попадают в цель.

В то время как этот метод возвращает правильные результаты, его производительность может быть неудовлетворительной. Если по сравнению с ограничивающим окном площадь представляющей интерес области небольшая, то большинство наших дротиков не будут попадать в цель. Описание эффективных алгоритмов для других специальных распределений, включая гауссово распределение, приводится далее — в *подразделе «Реализации»*.

Однако будьте осторожны с изобретением своих собственных методов, поскольку получить правильное распределение вероятностей очень трудно. Например, генерирование полярных координат посредством случайного выбора с равномерным распределением угла между 0 и $2\pi$ и смещением от 0 до $r$ представляет собой пример неправильного способа выбора равномерно распределенных точек из круга радиусом $r$. В этом случае половина сгенерированных точек будет располагаться на расстоянии $r/2$ от центра, в то время как здесь должна находиться только одна четвертая часть их количества! Эта разница достаточно большая, чтобы серьезно исказить результат, и в то же самое время настолько тонкая, что такие искажения легко не заметить.

- *Сколько времени нужно выполнять моделирование по методу Монте-Карло?*

  Чем больше времени выполняется программа моделирования, тем точнее *должна* быть аппроксимация предельного распределения. Но это происходит только до тех пор, пока не будет превышен *период* (длина цикла) генератора случайных чисел, после чего последовательность случайных чисел начнет повторяться, и дальнейшее исполнение программы не даст никакой дополнительной информации.

  Вместо того чтобы устанавливать максимальное значение периода, стоит выполнить программу моделирования с более коротким периодом, но несколько раз (от 10 до 100) и с разными начальными числами, а потом рассмотреть полученный диапазон результатов. Разброс значений даст вам хорошее представление о степени повторяемости полученных результатов. Так вы сможете избавиться от заблуждения, будто моделирование дает «правильный» ответ.

### Реализации

Отличный материал по генерированию случайных чисел и стохастическому моделированию можно найти на веб-сайте **https://www.agner.org/random**. Там даются ссылки на научные работы и приводится множество реализаций генераторов случайных чисел.

Параллельное моделирование предъявляет особые требования к генераторам случайных чисел. Например, как можно гарантировать независимость случайных потоков на

каждой машине? Одно из решений — воспользоваться объектно-ориентированными генераторами, описанными в работе [LSCK02], с длиной периода около $2^{191}$. Реализации этих генераторов на языках C, C++ и Java можно загрузить с веб-сайта **http://www.iro.umontreal.ca/~lecuyer/myftp/streams00**. Для работы на многопроцессорных компьютерах поддерживается генерирование независимых потоков случайных чисел. Другой подход — использовать библиотеку SPRNG (см. [MS00]), которую можно загрузить с веб-сайта **http://sprng.cs.fsu.edu**.

В Национальном институте стандартов США был разработан широкий набор статистических тестов для проверки генераторов случайных чисел (см. [BRS+10]). Это программное обеспечение и описывающий его доклад можно загрузить с веб-сайта **https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software**.

Генераторы действительно случайных чисел используют численные представления какого-либо физического процесса. Услуги по предоставлению случайных чисел, сгенерированных таким образом, предоставляются на веб-сайте **http://www.random.org**. Числа там генерируются на основе атмосферных шумов и проходят проверку статистическими тестами Национального института стандартов и технологий США. Это удачное решение, если вам требуется небольшое количество действительно случайных чисел — например, для проведения лотереи, а не сам генератор таких чисел.

#### *Примечания*

Генерирование случайных чисел очень подробно описано в книге Дональда Кнута [Knu97b]. В ней излагаются теоретические основы нескольких методов генерирования случайных чисел, включая метод серединных квадратов и регистр сдвига, которые не были рассмотрены здесь. Кроме того, подробно обсуждаются статистические тесты для проверки генераторов случайных чисел.

Информация о последних разработках в области генерирования случайных чисел собрана в книгах [Gen06] и [L'E12]. Генератор случайных чисел, называемый *вихрем Мерсенна* (Mersenne twister, см. [MN98]), имеет период длиной в $2^{19937} - 1$. Другие современные методы генерирования случайных чисел основаны на рекурсии (см. [Den05] и [PLM06]). Обзор методов генерирования случайных чисел с неравномерным распределением представлен в книге [HLD04], а в книге [PM88] — сравнение практического применения разных генераторов случайных чисел.

В прежние времена, когда компьютеры не были так распространены, в большинстве математических учебников печатались таблицы случайных чисел. Наиболее известным является [RC55], в котором приводится один миллион случайных чисел. Если хотите от души посмеяться, я рекомендую почитать сотни обзоров на эту книгу на Amazon (**https://www.amazon.com/Million-Random-Digits-Normal-Deviates**). Поразительно, сколько же все-таки шутников в Интернете.

Глубинная взаимосвязь между случайностью и информацией исследуется в теории *колмогоровской сложности*, согласно которой о сложности строки можно судить по ее сжимаемости. Строки истинно случайных символов несжимаемы. В соответствии с этой теорией кажущиеся случайными цифры значения $\pi$ в действительности не являются случайными, поскольку вся их последовательность определяется программой, реализующей разложение в ряд для $\pi$. Всестороннее рассмотрение теории колмогоровской сложности можно найти в книге [LV97].

### Родственные задачи

Условная и безусловная оптимизация (см. *разд. 16.5*), генерирование перестановок (см. *разд. 17.4*), генерирование подмножеств (см. *разд. 17.5*).

## 16.8. Разложение на множители и проверка чисел на простоту

**Вход.** Целое число $n$.

**Задача.** Определить, является ли целое число $n$ простым, а если нет, то найти его множители (рис. 16.8).

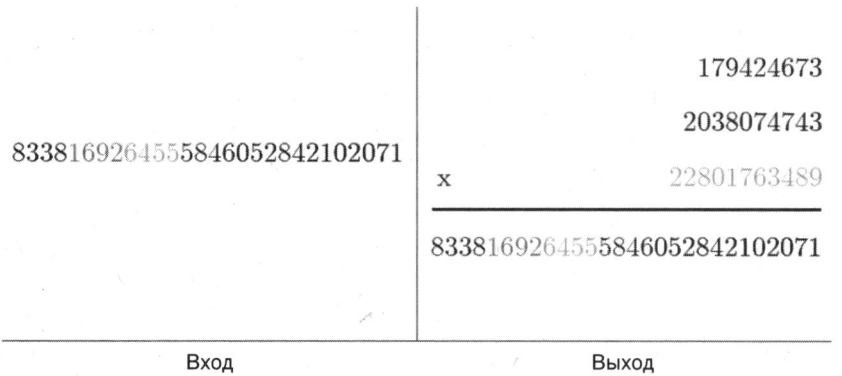



**Рис. 16.8.** Разложение на множители

### Обсуждение

Родственные задачи проверки целого числа на простоту и разложения его на множители имеют неожиданно много приложений, если учесть, что долгое время они считались представляющими только математический интерес. В частности, безопасность криптографической системы с открытым ключом RSA (см. *разд. 21.6*) основана на вычислительной неосуществимости решения задачи разложения на множители больших целых чисел. В связи с этим известно, что производительность хеш-таблиц обычно повышается, если их размер равен простому целому числу. Для этого процедура инициализации хеш-таблицы должна определить простое целое число, близкое к требуемому размеру хеш-таблицы. Наконец, с простыми числами просто интересно экспериментировать. Этим и объясняется распространенная в прошлом на UNIX-системах практика хранения программ для генерирования больших простых чисел в папке для игр.

Хотя задача разложения на множители и задача проверки числа на простоту значительно различаются в алгоритмическом отношении, они на самом деле близкородственные. Существуют алгоритмы, которые могут показать, что целое число является *составным* (т. е. не простым), не предоставляя при этом самих множителей. Чтобы убедиться в наличии таких алгоритмов, обратите внимание на тот факт, что можно продемонстрировать составную природу любого нетривиального целого числа, заканчивающегося на 0, 2, 4, 5, 6 или 8, не выполняя для этого самой операции деления.

Самым простым алгоритмом для решения обеих задач является проверка делением. Для этого выполняется деление $n/i$ для всех $1 < i \le \sqrt{n}$. Полученные при этом простые множители числа $n$ будут содержать, по крайней мере, один экземпляр такого делителя $i$, для которого $n/i = \lfloor n/i \rfloor$, если, конечно, $n$ не является простым числом. Но при этом необходимо обеспечить корректную обработку повторяющихся множителей, а также учесть все простые числа, большие $\sqrt{n}$.

Работу таких алгоритмов можно ускорить, используя таблицы заранее вычисленных простых чисел, чтобы не проверять все возможные значения $i$. Применение битовых векторов (см. *разд. 15.5*) позволяет представить удивительно большое количество простых чисел в неожиданно малом объеме памяти. Для хранения битового вектора, содержащего все нечетные числа до миллиона, требуется меньше чем 64 Кбайт. Еще более плотную упаковку можно получить, удалив все числа, кратные трем и другим небольшим простым числам.

Хотя алгоритм проверки делением исполняется за время $O(\sqrt{n})$, он *не* является полиномиальным алгоритмом по той причине, что для представления числа $n$ требуется только $\lg n$ битов, поэтому время исполнения алгоритма делением экспоненциально зависит от размера входа. Существуют значительно более быстрые (но с тем же экспоненциальным временем исполнения) алгоритмы, правильность которых основана на теории чисел. Самый быстрый алгоритм, названный *решетом числового поля* (number field sieve), использует случайную последовательность для создания системы конгруэнций, решение которой обычно дает делитель целого числа. С помощью этого метода было произведено разложение на множители целых чисел длиной в 250 цифр (829 бит), хотя для такой работы потребовалось выполнить громаднейший объем вычислений.

Проверку целых чисел на простоту значительно легче выполнять с помощью рандомизированных алгоритмов. Малая теорема Ферма утверждает, что

$$a^{n-1} = 1 \pmod{n}$$

для всех $a$, не делящихся на $n$, при условии, что $n$ является простым числом. Возьмем случайное значение $1 \le a \le n$ и вычислим остаток $a^{n-1} \pmod{n}$. Если этот остаток не равен 1, то это доказывает, что $n$ должно быть составным числом. Такие рандомизированные проверки чисел на простоту очень эффективны. С их помощью программа шифрования PGP (см. *разд. 21.6*) за несколько минут находит простые числа длиной в триста с лишним цифр для использования в качестве ключей шифрования.

Хотя кажется, что простые числа разбросаны среди целых чисел в случайном порядке, их распределение имеет определенную регулярность. *Теорема простых чисел* утверждает, что количество простых чисел, меньших чем $n$ (обычно обозначаемое как $\pi(n)$), приблизительно равно $n/\ln n$. Кроме этого, между простыми числами никогда нет больших промежутков, поэтому, вообще говоря, можно ожидать, что для того, чтобы найти первое простое число, большее чем $n$, надо будет проверить $\ln n$ целых чисел. Такое распределение и наличие быстрого рандомизированного теста на простоту объясняет, почему PGP находит большие простые числа со столь высокой скоростью.

Квантовые компьютеры теоретически способны очень быстро раскладывать на множители большие целые числа — и даже не очень быстро, а экспоненциально быстрее, чем

обычные компьютеры. Для меня не станет шоком, если к тому времени, когда я приступлю к работе над четвертым изданием этой книги, квантовые компьютеры будут способны выполнять эту операцию на практике. Однако на пути к этому существуют серьезные технические вызовы. В частности, для разложения на множители целых чисел масштаба RSA-шифрования (2048 битов) посредством алгоритма Шора требуется 20 миллионов кубитов — из-за необходимости выполнять обширные операции по исправлению ошибок (см. [GE19]).

Приложения квантовых вычислений, способные выполнять разложение на множители целых чисел масштаба RSA-шифрования, иногда называют *самоубийственными*, потому что сразу же после их первого успешного применения использование RSA-шифрования прекратится и такие приложения больше не понадобятся.

### Реализации

Существует несколько систем общего назначения для решения задач из вычислительной теории чисел. Система компьютерной алгебры PARI может решать сложные задачи теории чисел с целыми числами произвольной точности (чтобы быть точным — ограниченными длиной в 80 807 123 цифры на 32-разрядных машинах), а также с действительными, рациональными, сложными и алгебраическими числами и матрицами. Основа системы написана на языке C, а внутренние циклы для основных компьютерных архитектур — на ассемблере, и содержит свыше 200 специализированных математических функций. Систему PARI можно использовать в виде библиотеки, но она также имеет режим калькулятора, предоставляющий немедленный доступ ко всем типам и функциям. Загрузить систему можно с веб-сайта **http://pari.math.u-bordeaux.fr/**.

Высокопроизводительная переносимая библиотека NTL на языке C++ предоставляет структуры данных и алгоритмы для манипулирования целыми числами произвольной длины со знаком, а также векторов, матриц и многочленов над полем целых чисел и над конечными полями. Библиотеку NTL можно загрузить с веб-сайта **http://www.shoup.net/ntl/**.

Наконец, библиотека MIRACL, написанная на C/C++, реализует шесть разных алгоритмов для разложения целых чисел на множители, включая алгоритм квадратичного решета. Библиотеку MIRACL можно загрузить с веб-сайта **https://github.com/miracl/MIRACL**.

#### *Примечания*

Книги [CP05] и [Yan03] содержат описание современных алгоритмов проверки целых чисел на простоту и разложения на множители. Более общие сведения по вычислительной теории чисел можно найти в книгах [BS96] и [Sho09].

Агравал (Agrawal), Каял (Kayal) и Саксена (Saxena) предоставили первый детерминистский алгоритм с полиномиальным временем исполнения, проверяющий, является ли то или иное целое число составным (см. [AKS04]). При разработке своего незамысловатого алгоритма они выполнили тщательный анализ известных рандомизированных алгоритмов. Существование этого алгоритма является укором исследователям (таким как я), которые побаиваются заниматься классическими нерешенными задачами. Независимая трактовка полученного указанными разработчиками результата приведена в книге [Die04].

Класс сложности *co-NP* охватывает множество задач, нерешаемость экземпляров которых можно проверить с помощью алгоритма полиномиальной временной сложности, — при

условии наличия соответствующего сертификата. В теории сложности вычислений важное место занимает задача проверки действительности утверждения P = NP ∩ co-NP. Задача разрешимости «является ли $n$ составным числом» когда-то была наилучшим кандидатом для контрпримера. Членство $n$ в NP тривиально доказывается демонстрацией всех его множителей. А членом co-NP оно должно быть потому, что можно очень быстро доказать, что оно является простым (см. [Pra75]). Но эта цепочка логических рассуждений была разорвана недавним доказательством того, что задача проверки составных чисел является членом P. Дополнительная информация по классам сложности предоставляется в [AB09] и [GJ79].

Большой интерес в квантовых вычислениях возбудил алгоритм Шора для разложения целых чисел на множители за полиномиальное время [Sho99]. На момент подготовки этой книги разбиение на множители $15 = 3 \times 5$ остается на экспериментальном уровне текущего развития науки и техники (см. [MNM+16]). Введение в квантовые вычисления дается среди прочих в [Aar13] и [Ber19].

Рандомизированный алгоритм Миллера — Рабина (Miller — Rabin algorithm) для проверки (см. [Mil76], [Rab80]) целых чисел на простоту решает проблему чисел Кармайкла (Carmichael numbers), которые являются составными целыми числами, удовлетворяющими условиям теоремы Ферма. Самые лучшие алгоритмы для разложения целых чисел на множители используют метод квадратичного решета (см. [Poni84]) и метод эллиптической кривой (см. [Len87b]).

Механические вычислительные устройства задолго до наступления компьютерной эры предоставляли самый быстрый способ разложения целых чисел на множители. Занимательная история одного из таких устройств, созданного во время Первой мировой войны, изложена в работе [SWM95]. Приводимое в действие вручную вращением рукоятки устройство доказывало простоту числа $2^{31} - 1$ за 15 минут работы.

Число из задачи RSA-129 было разложено на множители в апреле 1994 года после восьми месяцев вычислений на 1600 компьютерах. Это событие оказалось особенно примечательным ввиду того обстоятельства, что в исходной работе RSA (см. [RSA78]) предполагалось, что решение этой задачи займет 40 квадрильонов лет (используя технологию 1970-х годов). Текущим рекордом по разложению на множители является успешная атака на число из задачи RSA-200, осуществленная в феврале 2020 года

### Родственные задачи

Криптография (см. *разд. 21.6*), арифметические операции высокой точности (см. *разд. 16.9*).

## 16.9. Арифметика произвольной точности

**Вход.** Два очень больших целых числа $x$ и $y$.

**Задача.** Вычислить $x + y$, $x - y$, $x \times y$ и $x/y$ (рис. 16.9).

### Обсуждение

Любой язык программирования, имеющий уровень выше ассемблера, поддерживает арифметические операции сложения, вычитания, умножения и деления с целыми и действительными числами с одинарной и, возможно, двойной точностью. А что если мы захотим выразить государственный долг США в центах? Для выражения количества центов стоимостью в 22,4 триллиона долларов потребуется число из 16 десятичных

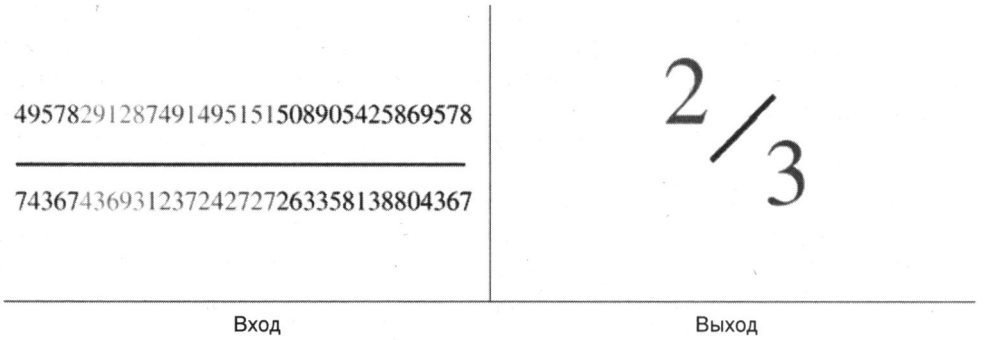



**Рис. 16.9.** Деление очень больших целых чисел

цифр, что намного больше, чем может вместиться в 32-битовое компьютерное слово (но в 64-битовое поместится без каких бы то ни было проблем).

В других приложениях могут возникнуть *намного* бóльшие целые числа. Например, для обеспечения достаточной безопасности в алгоритме RSA для криптографии с открытым ключом рекомендуется использовать целочисленные ключи длиной не менее 2048 битов (или, эквивалентно, 617 цифр). Исследовательские эксперименты в области теории чисел также требуют выполнения операций с большими числами. Однажды я решил незначительную нерешенную задачу, выполнив точные вычисления, результат которых представлен целым числом $\binom{5906}{2953} \approx 9{,}93285 \cdot 10^{1775}$, как показано в *разд. 6.9*.

Прежде чем приступить к работе с очень большими целыми числами, ответьте на следующие вопросы:

- *Я решаю экземпляр задачи, требующей больших целых чисел, или имею дело со встроенным приложением?*

  Если вам просто нужен ответ для конкретного вычисления с большими целыми числами, как в моем случае с нерешенной задачей, упомянутом ранее, то следует рассмотреть использование интерпретатора Python или системы компьютерной алгебры, такой как Maple или Mathematica. Эти средства по умолчанию предоставляют возможности арифметических вычислений произвольной точности и используют для интерфейса удобные языковые интерпретаторы. В таком случае решение вашей задачи часто будет заключаться в программе из 5–10 строчек.

  Если же вы работаете со встроенным приложением, требующим выполнения арифметических операций с высокой точностью, то следует использовать какую-либо математическую библиотеку, поддерживающую вычисления с произвольной точностью. Эти библиотеки в дополнение к четырем основным операциям часто предоставляют дополнительные функции для таких вычислений, как наибольший общий делитель и квадратные корни. Подробности приведены далее — в *подразделе «Реализации»*.

- *Какой уровень точности требуется?*

  Иными словами, имеется ли верхняя граница для чисел, с которыми вы работаете, или же требуется неограниченная произвольная точность представления. От этого

зависит, сможете ли вы использовать для представления целых чисел массив фиксированной длины или вам потребуется применение связных списков. Массив проще для реализации и работы, и в большинстве приложений он не создает никаких ограничений.

- *Какое основание следует использовать?*

  Возможно, что легче всего будет реализовать свой собственный пакет для выполнения арифметических операций с высокой точностью в десятичной системе, представляя каждое целое число в виде строки цифр с основанием 10. Но с точки зрения вычислений гораздо эффективнее использовать в качестве основания большое число, в идеале равное квадратному корню из наибольшего целого числа, поддерживаемого аппаратно.

  Почему? Потому, что чем больше основание, тем меньше требуется цифр для представления чисел. Например, сравните количество цифр в одном и том же значении, но представленном в десятичной и двоичной системах счисления: 64 и 1 000 000 соответственно. Так как аппаратная операция сложения обычно выполняется за один тактовый цикл независимо от фактических значений суммируемых чисел, то наилучшая производительность достигается при использовании максимально возможного основания. Верхняя граница основания определяется значением $b = \sqrt{\max \text{int}}$, что позволяет избежать арифметического переполнения при умножении двух таких «цифр».

  Основной сложностью использования большого основания является необходимость преобразовывать целые числа в представление с основанием 10 для входа и выхода. Но это преобразование легко выполняется применением всех четырех арифметических операций с высокой точностью.

- *Какой точности будет достаточно?*

  Сложение аппаратными средствами выполняется намного быстрее, чем программными, поэтому использование арифметических операций с высокой точностью в ситуациях, когда такая точность не требуется, значительно снижает производительность. Арифметические вычисления с высокой точностью относятся к тем немногочисленным задачам, для которых реализация внутренних циклов на ассемблере оказывается хорошим способом ускорения работы программы. Точно так же наложение маски на уровне битов и использование операций сдвига вместо арифметических операций может ускорить выполнение программы — но для этого вы должны хорошо разбираться в особенностях машинного представления целых чисел.

Далее приводятся оптимальные алгоритмы для основных арифметических операций:

- *сложение.*

  Простой школьный способ сложения, при котором числа выравниваются по десятичному разделителю, после чего их цифры складываются с переносом справа налево, имеет время исполнения, линейно зависящее от количества цифр. Существуют более сложные параллельные алгоритмы с предсказанием переноса (carry-look-ahead) для реализации низкоуровневого машинного сложения. Скорее всего, они

используются в вашем микропроцессоре для выполнения операций сложения с низкой точностью;

- *вычитание.*

  Изменяя знаковые биты, мы можем превратить операцию вычитания в специальный случай сложения: $(A - (-B)) = (A + B)$. Сложной частью операции вычитания является «заем» из вышестоящего разряда. Для упрощения можно всегда выполнять вычитание меньшего числа из числа с бóльшим абсолютным значением и корректировать знаки позже. Тогда у нас всегда будет, откуда взять «заем». Для упрощения вычитания целых чисел со знаком разработчики компьютерных архитектур используют числовые представления в дополнительном двоичном коде;

- *умножение.*

  На больших целых числах выполнение операции умножения повторяющимися операциями сложения занимает экспоненциальное время, поэтому избегайте использования этого метода. Поцифровое умножение школьным методом достаточно легко поддается программированию и исполняется за время $O(n^2)$ при умножении двух $n$-разрядных целых чисел. Для очень больших целых чисел рекомендуется алгоритм Карацубы (Karatsuba's algorithm) типа «разделяй и властвуй» с временем исполнения $O(n^{1,59})$. Дэн Грейсон (Dan Grayson), разработчик арифметических операций произвольной точности системы компьютерной алгебры Mathematica, обнаружил, что точка перехода к этому алгоритму находится среди чисел, имеющих гораздо менее ста цифр. Еще более быстрым является алгоритм на основе преобразований Фурье. Такие алгоритмы рассматриваются в *разд. 16.11*;

- *деление.*

  Как и в случае с умножением, выполнение деления посредством многократного вычитания занимает экспоненциальное время, поэтому наиболее приемлемым алгоритмом будет «деление в столбик», которому нас учат в школе. Это требует выполнения операции умножения с произвольной точностью и операции вычитания в качестве вызываемых процедур, а также определения правильной цифры в каждой позиции частного методом проб и ошибок.

  Операцию деления целых чисел можно свести к операции умножения, хотя такое сведение далеко не тривиально. Так что, если вы разрабатываете асимптотически быстрое умножение, то результат можно будет использовать и для реализации операции деления. Соответствующие подробности излагаются в указанных далее источниках;

- *возведение в степень.*

  Значение $a^n$ можно вычислить, выполнив $n - 1$ умножений числа $a$ на само себя, — т. е. $a \times a \times ... \times a$. Но существует намного лучший алгоритм типа «разделяй и властвуй», основанный на том обстоятельстве, что $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. Если число $n$ четное, то тогда $a^n = (a^{n/2})^2$. А если $n$ нечетное, то тогда $n^2 = a\left(a^{\lfloor n/2 \rfloor}\right)^2$. В любом случае размер показателя степени был уменьшен наполовину, что обошлось нам, самое большее, в две операции умножения. Таким образом, для вычисления конечного значе-

ния будет достаточно $O(\lg n)$ операций умножения. Псевдокод соответствующего алгоритма показан в листинге 16.2.

**Листинг 16.2. Алгоритм возведения в степень**

```
function power(a, n)
   if (n = 0) return(1)
   x = power(a, ⌊n/2⌋)
   if (n is even) then return(n2)
      else return(a × x2)
```

Арифметические операции высокой, но не произвольной точности, удобно выполнять, используя *китайскую теорему об остатках* и модульную арифметику. Китайская теорема об остатках утверждает, что каждое целое число в диапазоне от 1 до $P = \prod_{i=1}^{k} p_i$ однозначно определяется его набором остатков от деления $p_i$, где каждая пара $p_i$, $p_j$ состоит из взаимно простых целых чисел. С помощью таких систем остаточных классов можно выполнять операции сложения, вычитания и умножения (но не деления), причем для манипуляций с большими целыми числами не потребуются сложные структуры данных.

Многие алгоритмы для вычислений с большими целыми числами можно прямо использовать для вычислений с многочленами. Особенно полезным алгоритмом для быстрой оценки многочленов является правило Горнера. Если выражение $P(x) = \sum_{i=0}^{n} c_i \cdot x^i$ вычислять почленно, то нужно будет выполнить $O(n^2)$ операций умножения. Намного лучше воспользоваться тем обстоятельством, что $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + ...)))$. Вычисление этого выражения требует только линейного количества операций.

### Реализации

Язык программирования Python и коммерческие системы компьютерной алгебры типа Maple и Mathematica поддерживают арифметические операции с высокой точностью. Для быстрых результатов в независимом приложении лучше всего воспользоваться одной из этих систем, если у вас имеется такая возможность. В остальном материале этого подраздела рассматриваются реализации для встроенных приложений.

Ведущей библиотекой на языке C/C++ для быстрого выполнения арифметических операций с высокой точностью является библиотека GMP (GNU Multiple Precision Arithmetic Library — библиотека GNU для арифметических операций с многократно увеличенной точностью), которая поддерживает операции с целыми числами со знаком, рациональными числами и числами с плавающей запятой. Загрузить ее можно с веб-сайта **http://gmplib.org**.

Класс `BigInteer` пакета java.math предоставляет аналоги операторов с произвольной точностью для всех элементарных операторов Java. Этот класс также содержит дополнительные операции для модульной арифметики, вычисления наибольшего общего делителя, проверки на простоту, генерирования простых чисел, манипулирования битами и других операций.

Менее производительная и не так хорошо протестированная моя собственная реализация арифметических операций с высокой точностью содержится в библиотеке из моей книги (см. [SR03]). Подробности см. в *разд. 22.1.9*.

Существует несколько общих систем для вычислительной теории чисел, каждая из которых поддерживает операции с целыми числами с произвольной точностью. Информацию о библиотеках PARI и NTL для работы с задачами теории чисел можно найти в *разд. 16.8*.

#### Примечания

Основным справочником по алгоритмам для всех основных арифметических операций, включая их реализацию на языке ассемблера MIX, является книга Дональда Кнута [Knu97b]. Более современное описание вычислительной теории чисел представлено в книгах [BS96] и [Sho09]. В работе [BZ10] предоставляется современное трактование зрелой темы компьютерной арифметики.

В число работ с описанием алгоритма типа «разделяй и властвуй» для умножения с временем исполнения $O(n^{1,59})$ входят книги [AHU74] и [Man89]. Алгоритм на основе быстрого преобразования Фурье умножает два числа длиной в $n$ битов за время $O(n \lg n \lg \lg n)$ (см. [SS71]). Его описание можно найти, например, в [AHU74] и [Knu97b]. Здесь же приводится описание сведения задачи деления целых чисел к задаче их умножения. В 2019 году эта производительность была, наконец, улучшена до $O(n \lg n)$ (см. [HVDH19] и [HVDHL16]). Поразительно, что для разработки асимптотически оптимального алгоритма для такой фундаментальной задачи, как умножение целых чисел, потребовалось так много времени. Использование быстрого умножения для выполнения других арифметических операций обсуждается в работе [Ber04].

Хорошие описания алгоритмов модульной арифметики и китайской теоремы об остатках содержатся в [AHU74] и [CLRS09]. А в книге [CLRS09] дано описание алгоритмов для выполнения элементарных арифметических операций.

Самым старым представляющим интерес алгоритмом, вероятно, является алгоритм Евклида для вычисления наибольшего общего делителя двух чисел. Его описание можно найти, например, в [Knu97b] и [CLRS09].

### Родственные задачи

Разложение целых чисел на множители (см. *разд. 16.8*), криптография (см. *разд. 21.6*).

## 16.10. Задача о рюкзаке

**Вход.** Множество предметов $S = \{1, ..., n\}$, где размер $i$-го предмета равен $s_i$, а значение — $v_i$. Емкость рюкзака равна $C$.

**Задача.** Найти подмножество $S'$ множества $S$, которое максимизирует значение $\sum_{i \in S'} v_i$, учитывая, что $\sum_{i \in S'} s_i \le C$, т. е. что все предметы помещаются в рюкзак размером $C$ (рис. 16.10).

### Обсуждение

Задача о рюкзаке возникает в ситуациях распределения ресурсов при наличии финансовых ограничений. Например, как выбрать вещи, которые нужно купить на фиксиро-

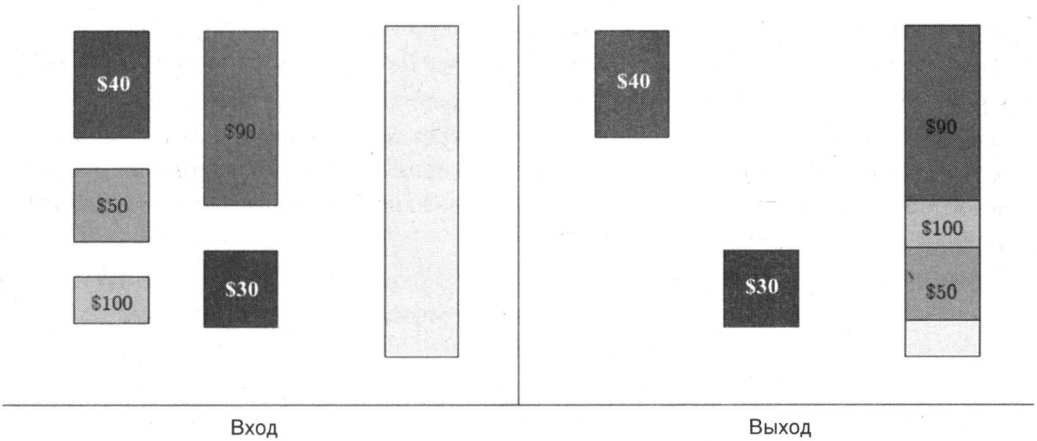



**Рис. 16.10.** Задача о рюкзаке

ванную сумму? Так как все предметы имеют свою стоимость и значимость, мы стремимся добиться максимальной значимости при заданной стоимости. Само название задачи — *задача о рюкзаке* — должно вызывать в мыслях образ туриста, который из-за ограниченности размера рюкзака старается положить в него самые нужные и компактные вещи.

Самая распространенная постановка задачи имеет ограничение вида «0/1», подразумевающее, что каждый предмет должен быть либо положен в рюкзак целиком, либо не положен вовсе. В большинстве реальных ситуаций предметы не получается разделять на части произвольным образом, поэтому нельзя взять только одну банку лимонада из шестибаночной упаковки или, вообще, часть содержимого одной банки. Вот это ограничение «0/1» и делает задачу о рюкзаке такой сложной, поскольку, когда возможность разделения предметов на части имеется, оптимальное решение находится «жадным» алгоритмом. Мы тогда просто вычисляем «стоимость килограмма» каждого предмета и вкладываем в рюкзак самый дорогой предмет или наибольшую его часть и повторяем эту операцию с самым дорогим предметом из оставшихся до тех пор, пока не заполним весь рюкзак. Но, к сожалению, в большинстве приложений ограничение «0/1» присутствует.

Приведем вопросы, которые возникают при выборе наилучшего алгоритма для решения задачи о рюкзаке.

- *Имеют ли все предметы одинаковую стоимость или, возможно, одинаковый размер?*

  Когда стоимость всех предметов одинаковая, то, чтобы получить наибольшую общую стоимость, мы просто берем максимальное количество предметов. Поэтому оптимальное решение здесь — отсортировать все предметы в возрастающем порядке по размеру, после чего вкладывать их в рюкзак в этом порядке до тех пор, пока имеется свободное место. Подобным же образом задача решается и в ситуации, когда все предметы одинакового размера, но разной стоимости. Тогда предметы сортируются в убывающем порядке по стоимости и в этом порядке и укладываются в рюкзак. Это легкие частные случаи задачи о рюкзаке.

- *Одинакова ли стоимость килограмма для всех предметов?*

  Если да, то в таком случае мы игнорируем цену и просто пытаемся минимизировать незаполненное пространство рюкзака. К сожалению, даже этот ограниченный случай задачи является NP-полным, поэтому нельзя ожидать найти эффективный алгоритм, который всегда выдает решение. Но не отчаивайтесь, поскольку задача о рюкзаке оказывается «легкой» сложной задачей, которую обычно можно решить с помощью представленных далее алгоритмов.

  Важным частным случаем варианта задачи о рюкзаке с одинаковой стоимостью килограмма каждого предмета является задача *разбиения множества целых чисел*, представленная графически на рис. 16.11.

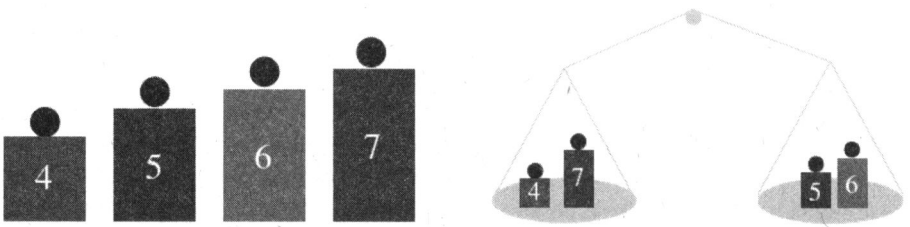



**Рис. 16.11.** Задача разбиения множества целых чисел — частный случай задачи о рюкзаке

Здесь нам нужно разделить элементы множества $S$ на два подмножества $A$ и $B$ таким образом, чтобы $\sum_{a\in A} a = \sum_{b\in B} b$ — или, в более общем смысле, чтобы разность их была минимальной. Задачу разбиения множества целых чисел можно рассматривать как задачу упаковки двух рюкзаков одинаковой вместимости или одного рюкзака с емкостью, вдвое меньшей чем исходная, поэтому все три задачи тесно связаны и являются NP-полными.

Вариант задачи с одинаковой стоимостью килограмма для всех элементов часто называется *задачей о сумме подмножества* (subset sum problem), поскольку мы стремимся найти такое подмножество элементов, общая стоимость которых будет равной определенному целевому значению $C$ — т. е. емкости нашего рюкзака.

- *Размеры всех предметов представлены относительно небольшими целыми числами?*

  Для варианта задачи, когда размеры предметов и емкость рюкзака представлены целыми числами, существует эффективный алгоритм поиска оптимального решения (см. *разд. 10.5*) с временной сложностью $O(nC)$ и сложностью по памяти $O(C)$. Подойдет ли этот алгоритм для решения вашей конкретной задачи, зависит от размера $C$. Алгоритм дает отличные результаты для $C \leq 1\,000$, но не очень хорошие для $C \geq 1\,000\,000\,000$.

  Алгоритм работает следующим образом. Пусть $S'$ — это набор элементов, а $C[i, S']$ будет иметь значение *истина* тогда и только тогда, когда существует подмножество множества $S'$, общий размер всех элементов которого равен точно $i$. Отсюда следует, что $C[i,\varnothing]$ имеет значение *ложь* для всех $1 \leq i \leq C$. Далее мы по одному добавляем элементы $s_j$ к $S'$ и обновляем затронутые значения $C[i, S']$. Обратите внимание,

что $C[i, S' \cup s_j]$ = *истина* тогда и только тогда, когда истинны $C[i, S']$ или $C[i - s_j, S']$, поскольку мы либо используем $s_j$ в получении суммы, либо нет. Мы определяем все суммы, которые можно получить, перебрав $n$ раз все элементы в $C$ — по одному разу для каждого $s_j$, где $1 \le j \le n$, обновляя таким образом массив. Решением задачи является индекс истинного элемента наибольшего размера. Чтобы воссоздать подмножество, дающее решение, для каждого $1 \le i \le C$, мы должны сохранять имя элемента, который изменяет значение $C[i]$ с *ложь* на *истина*, после чего перебрать элементы массива в обратном направлении.

Эта формулировка в стиле динамического программирования игнорирует значения элементов. Поэтому, чтобы обобщить этот алгоритм, в каждом элементе массива сохраняется значение наилучшего текущего подмножества, для которого сумма размеров элементов равна $i$. Теперь обновление выполняется, когда сумма стоимости $C[i - s_j, S']$ и стоимости $s_j$ лучше, чем предыдущая стоимость $C[i, S' \cup s_j]$.

- *Что делать, если имеется несколько рюкзаков?*

  В таком случае задачу лучше рассматривать как задачу разложения по контейнерам. Алгоритмы для решения задачи разложения по контейнерам и задачи раскроя (cutting-stock) описаны в *разд. 20.9*. А реализации алгоритмов решения задачи с несколькими рюкзаками рассматриваются далее — в *подразделе «Реализации»*.

Точные решения для рюкзаков большого объема можно найти с помощью целочисленного программирования или поиска с возвратом. Наличие или отсутствие элемента $i$ в оптимальном подмножестве обозначается целочисленной переменной $x_i$, принимающей значения 0 или 1. Мы максимизируем выражение $\sum_{i=1}^{n} x_i \cdot v_i$ при ограничивающем условии $\sum_{i=1}^{n} x_i \cdot s_i \le C$. Реализации алгоритмов целочисленного программирования рассматриваются в *разд. 16.6*.

Когда получение точного решения оказывается слишком дорогим в вычислительном отношении, возникает необходимость в использовании эвристических алгоритмов. Простой эвристический «жадный» алгоритм вкладывает предметы в рюкзак согласно ранее рассмотренному правилу максимальной стоимости килограмма. Часто такое эвристическое решение близко к оптимальному, но, в зависимости от конкретного экземпляра задачи, оно также может оказаться сколь угодно плохим. Правило стоимости килограмма можно использовать, чтобы уменьшить размер задачи в алгоритмах исчерпывающего перебора, чтобы в дальнейшем не рассматривать «дешевые, но тяжелые» предметы.

Другой эвристический алгоритм основан на *масштабировании*. Динамическое программирование хорошо подходит для тех случаев, когда емкость рюкзака выражена достаточно небольшим целым числом — скажем, $\le C_s$. А если нам приходится иметь дело с рюкзаком, емкость которого больше, чем это значение, — т. е. $C > C_s$? В таком случае мы можем уменьшить размеры всех элементов в $C/C_s$ раз, округлить полученные размеры до ближайшего целого числа, после чего использовать динамическое программирование с этими уменьшенными элементами. Масштабирование хорошо зарекомендовало себя на практике, особенно при небольшом разбросе размеров элементов.

### Реализации

Коллекцию реализаций алгоритмов на языке FORTRAN для разных версий задачи о рюкзаке можно загрузить с веб-сайта **http://www.or.deis.unibo.it/kp.html**. Здесь же можно загрузить электронную версию книги [MT90a].

Хорошо организованную коллекцию реализаций алгоритмов на языке C для решения разных видов задачи о рюкзаке и родственных вариантов задачи — таких как разложение по контейнерам и загрузка контейнера, можно загрузить с веб-сайта **http://www.diku.dk/~pisinger/codes.html**. Самый мощный код основан на алгоритме динамического программирования, описание которого дается в работе [MPT99].

Алгоритм 632 на языке FORTRAN из коллекции ACM предназначен для решения задачи о рюкзаке с ограничением вида «0/1». Кроме того, он поддерживает работу с несколькими рюкзаками. Подробности см. в *разд. 22.1.4*.

#### Примечания

Самым свежим справочником по задаче о рюкзаке и ее вариантах является книга [KPP04]. Книга [MT90a] и обзорная статья [MT87] представляют собой обычные справочные пособия по задаче о рюкзаке, содержащие как теоретические, так и экспериментальные результаты ее решения. Замечательное описание алгоритмов целочисленного программирования для решения задач о рюкзаке представлено в книге [MPT99]. Алгоритмы решения задачи о рюкзаке с ограничением вида «0/1» обсуждаются в работе [MPT00].

Аппроксимирующий метод дает решение, близкое к оптимальному, за время, зависящее полиномиально от размера входа и коэффициента аппроксимации $\varepsilon$. Такое очень строгое ограничение заставляет искать компромисс между временем исполнения и качеством аппроксимации. Хорошие описания аппроксимирующих методов с полиномиальным временем исполнения для решения задачи о рюкзаке и суммы подмножества можно найти в [IK75], [BvG99] и [CLRS09]. Аппроксимирующие алгоритмы с полиномиальным временем исполнения существуют даже для решения задачи нескольких рюкзаков. См., например, [CK05].

Интересный особый случай задачи о рюкзаке представляет задача 3SUM, в которой для заданных трех множеств $A$, $B$ и $C$, каждое из которых содержит $n$ целых чисел, требуется найти такие $a \in A$, $b \in B$ и $c \in C$, чтобы $a + b = c$. Наилучший известный алгоритм для решения этой задачи имеет временную сложность $O(n^2)$, и сведение к задаче 3SUM часто используется для неявного утверждения отсутствия решения с меньшей, чем квадратической, временной сложностью для определенной задачи (см. [GO95]).

Задача векторной упаковки по контейнерам представляет собой обобщенную задачу о рюкзаке, где емкость рюкзака ограничена по $d$ осям (скажем, характеристиками центрального процессора и памяти), а каждый объект определяется вектором из соответствующих $d$ требований. Результаты исследований эвристических алгоритмов для решения задачи векторной упаковки по контейнерам в контексте размещения виртуальных машин в центрах хранения и обработки данных излагаются в работе [PTUW11].

Первый алгоритм для общего метода шифрования с открытым ключом был основан на сложности задачи о рюкзаке. Описание см. в книге [Sch15].

### Родственные задачи

Разложение по контейнерам (см. *разд. 20.9*), целочисленное программирование (см. *разд. 16.6*).

# 16.11. Дискретное преобразование Фурье

**Вход.** Последовательность из $n$ действительных или комплексных значений $h_i$ функции $h$, выбранных через одинаковые интервалы.

**Задача.** Дискретное преобразование Фурье $H_m = \sum_{k=0}^{n-1} h_k e^{-2\pi ikm/n}$ для $0 \le m \le n-1$ (рис. 16.12).

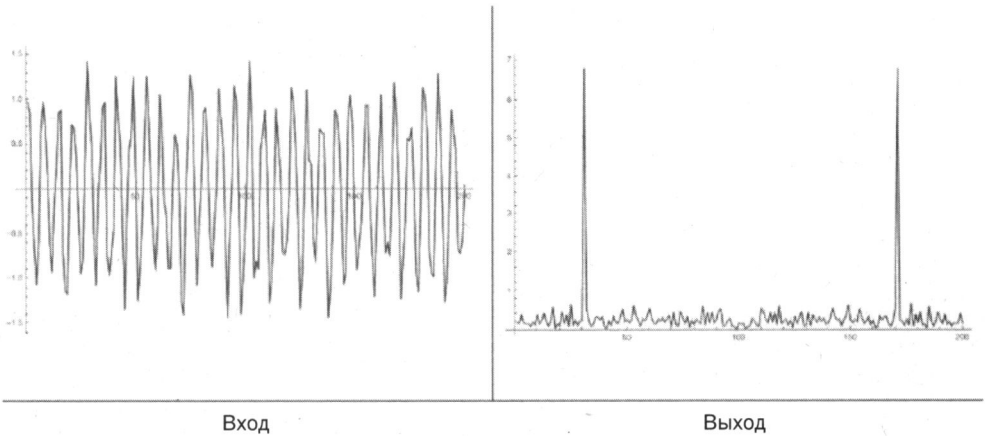



**Рис. 16.12.** Пример преобразования Фурье

### Обсуждение

В то время как программисты обычно плохо разбираются в преобразованиях Фурье, научные работники и инженеры обращаются с ними вполне свободно. В функциональном отношении преобразования Фурье предоставляют способ для преобразования выборок стандартного временного ряда в *частотную область*. При этом функция приобретает двойственное представление, и определенные операции становятся проще для выполнения, чем в исходной временной области. Преобразования Фурье находят следующие применения:

- *фильтрация.*

  Выполнение преобразования Фурье для функции равносильно представлению ее в виде суммы синусных составляющих. Устранив «лишние» высокочастотные и/или низкочастотные компоненты (т. е. исключив некоторые слагаемые), мы можем отфильтровать из изображения шум и другие нежелательные явления. Выполнив после этого обратное преобразование Фурье, мы возвратимся обратно во временную область. Например, всплески на графике, представленном на рис. 16.12, *справа*, соответствуют периоду одной синусной составляющей и моделируют входные данные. Остальные компоненты — это просто шум;

- *сжатие изображений.*

  Сглаженное и отфильтрованное изображение содержит меньший объем информации, чем исходное изображение, в то же самое время сохраняя похожий внешний

вид. Удалив компоненты, вносящие сравнительно небольшой вклад в изображение, мы уменьшаем размер изображения за счет незначительного понижения его точности;

- *свертка и обращение свертки.*

  Преобразования Фурье можно использовать для эффективного выполнения свертки двух последовательностей. *Сверткой* (convolution) называется попарное умножение элементов двух разных последовательностей — например, при перемножении двух многочленов $f$ и $g$ с $n$ переменными или при сравнении двух текстовых строк. Реализация такой операции напрямую имеет квадратичное время исполнения (т. е. $O(n^2)$), в то время как время исполнения алгоритма на основе быстрого преобразования Фурье равно $O(n \lg n)$.

  Приведем другой пример из области обработки изображений. Так как сканер измеряет уровень освещенности участка изображения, а не отдельной его точки, то отсканированное изображение всегда немного смазано. Исходный сигнал можно восстановить, выполнив обращение свертки входного сигнала посредством гауссовой функции рассеяния точки;

- *вычисление корреляционной функции.*

  Корреляционная функция двух функций $f(t)$ и $g(t)$ определяется как

$$z(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau.$$

  Эту функцию можно с легкостью вычислить, используя преобразования Фурье. Когда две функции $f(t)$ и $g(t)$ имеют похожую форму, но сдвинуты по отношению друг к другу (например, если функции $f(t) = \sin(t)$, а $g(t) = \cos(t)$), значение $z(t_0)$ будет большим для такого смещения $t_0$. В качестве примера практического применения допустим, что мы хотим определить, нет ли в нашем генераторе случайных чисел каких-либо нежелательных периодичных последовательностей. Для этого мы можем сгенерировать длинную последовательность случайных чисел, преобразовать их во временную последовательность (в которой $i$-е число соответствует моменту времени $i$), после чего вычислить корреляционную функцию этой последовательности. Любой большой всплеск будет индикатором возможной периодичности.

Дискретное преобразование Фурье принимает на входе $n$ комплексных чисел $h_k$, где $0 \le k \le n-1$, соответствующих равномерно распределенным точкам временной последовательности, и выдает на выходе $n$ комплексных чисел $H_k$, где $0 \le k \le n-1$, каждое из которых описывает синусоидальную функцию соответствующей частоты. Дискретное преобразование Фурье определяется следующей формулой:

$$H_m = \sum_{k=0}^{n-1} h_k \cdot e^{-2\pi ikm/n} = \sum_{k=0}^{n-1} h_k \left[ \cos\left(\frac{2\pi km}{n}\right) - i\sin\left(\frac{2\pi km}{n}\right) \right],$$

а обратное преобразование Фурье — следующей:

$$h_m = \sum_{k=0}^{n-1} H_k \cdot e^{2\pi ikm/n} = \frac{1}{n}\sum_{k=0}^{n-1} H_k \left[ \cos\left(\frac{2\pi km}{n}\right) + i\sin\left(\frac{2\pi km}{n}\right) \right].$$

Таким образом мы можем с легкостью перемещаться между $h$ и $H$.

Поскольку выход дискретного преобразования Фурье состоит из $n$ чисел, каждое из которых вычисляется по формуле, содержащей $n$ элементов, то его время исполнения равно $O(n^2)$. А алгоритм быстрого преобразования Фурье вычисляет дискретное преобразование Фурье за время $O(n \log n)$. Вероятно, это самый важный известный алгоритм, т. к. он лежит в основе всей современной обработки сигналов. Под общим названием быстрого преобразования Фурье известно несколько алгоритмов, использующих метод «разделяй и властвуй». По сути, задача вычисления дискретного преобразования Фурье по $n$ точкам сводится к вычислению двух преобразований, каждое по $n/2$ точкам, которые потом применяются рекурсивно.

Алгоритм быстрого преобразования Фурье обычно предполагает, что число $n$ является степенью двойки. Если ваше значение $n$ не отвечает этому условию, то будет лучше дополнить данные нулями, чтобы создать $n = 2^k$ элементов, чем искать более общий код.

Многие системы обработки сигналов имеют строгие условия работы в реальном времени, поэтому алгоритмы быстрого преобразования Фурье часто реализуются аппаратно или, по крайней мере, на языке ассемблера, настроенного под конкретную машину. Учтите это обстоятельство, если используемый вами код из упомянутых далее источников окажется слишком медленным.

### Реализации

На первом месте среди открытых кодов алгоритма быстрого преобразования Фурье стоит библиотека FFTW. Это библиотека процедур на языке C для вычисления дискретного преобразования Фурье в одном или нескольких измерениях, поддерживающая вход произвольного размера и данные как действительного, так и комплексного типа. Результаты обширного тестирования доказывают, что это действительно самое быстрое известное преобразование Фурье. В 1999 году библиотеке был присужден приз Дж. Х. Вилкинсона в области численного программного обеспечения (J. H. Wilkinson Prize for Numerical Software). Дополнительную информацию вы найдете на вебсайте **http://www.fftw.org/**.

Пакет FFTPACK содержит процедуры на языке FORTRAN для вычисления быстрого преобразования Фурье периодических и других симметрических последовательностей. Пакет включает комплексные, действительные, синусоидальные и четвертьволновые преобразования. Загрузить его можно с веб-сайта **http://www.netlib.org/fftpack**. Научная библиотека GNU для C/C++ предоставляет версию библиотеки FFTPACK. Подробности см. по адресу **http://www.gnu.org/software/gsl/**.

#### *Примечания*

Хорошим введением в предмет преобразований Фурье и быстрых преобразований Фурье являются книги [Bra99] и [Bri88]. Описание преобразования можно найти в книге [PFTV07]. Изобретение быстрого преобразования Фурье обычно приписывают Кули (Cooley) и Туки (Tukey) — см. [CT65]. Подробную историю вопроса можно найти в [Bri88].

Нечувствительный к кэшированию алгоритм быстрого преобразования Фурье приведен в докладе [FLPR99]. Да и само понятие нечувствительных к кэшированию алгоритмов было впервые предложено в нем. Библиотека FFTW основана на этом алгоритме. Подробности устройства библиотеки FFTW см. в [FJ05]. Для частного случая задачи, когда тре-

буется определить только $k$ наибольших коэффициентов трансформации при $k<<n$, существуют более эффективные алгоритмы — см. [HIKP12].

Интересный алгоритм типа «разделяй и властвуй» для умножения многочленов (см. [KO63]) с временем исполнения $O(n^{1.59})$ рассматривается в книгах [AHU74] и [Man89]. Алгоритм на основе быстрого преобразования Фурье, умножающий два числа длиной в $n$ битов за время $O(n \lg n \lg \lg n)$, был разработан Шонхаге (Schonhage) и Штрассеном (Strassen). Он представлен в [SS71].

*Квантовое преобразование Фурье* предоставляет экспоненциальное ускорение по сравнению с классическим вариантом, обрабатывая $2^n$ амплитуд, хранящихся в $n$ кубитах, и выполняя при этом всего лишь $O(n^2)$ операций. Подробности см. в [NC02]. Но для достижения такого результата требуется эффективное размещение (или извлечение) требуемых амплитуд в кубитах. Указанная операция является основным компонентом квантового алгоритма Шора для разложения на множители (см. [Sho99]). На интуитивном уровне это можно объяснить следующим образом: рассмотрим множество синусоидальных функций с периодами 2, 3, 5, 7, 11, ... в частотной области. Если $n$ делится на любое из этих значений, тогда $n$ должно иметь пик во временной области.

Вопрос о том, действительно ли комплексные переменные являются необходимыми в быстрых алгоритмах для выполнения свертки, остается открытым. К счастью, в приложениях быструю свертку обычно можно использовать в виде «черного ящика». Быстрая свертка лежит в основе многих разновидностей алгоритмов для сравнения строк (см. [Ind98]).

В последнее время было предложено вместо преобразования Фурье использовать в фильтрации специальные функции, называемые *вейвлетами*. Введение в эту тему можно найти в [BN09].

### Родственные задачи

Сжатие данных (см. *разд. 21.5*), арифметические операции с высокой точностью (см. *разд. 16.9*).

# ГЛАВА 17

# Комбинаторные задачи

В этой главе мы рассмотрим несколько алгоритмических задач чисто комбинаторного характера. В их число входят задача сортировки и задача поиска, которые были первыми нечисловыми задачами, решенными с помощью электронных вычислительных машин. Сортировку можно рассматривать как полное упорядочивание ключей, а поиск и выбор — как идентификацию ключей по их положению в упорядоченной последовательности.

Определенное внимание мы также уделим и другим комбинаторным объектам: перестановкам, разбиениям, подмножествам, календарям и расписаниям. Особый интерес представляют алгоритмы, которые ранжируют комбинаторные объекты, — т. е. устанавливают соответствие между объектом и уникальным целым числом, или выполняют обратную операцию, возвращая объект по числу. Операции ранжирования упрощают многие другие задачи — например, генерирование случайных объектов (выбирается произвольное число и выполняется операция, обратная ранжированию) или перечисление всех объектов по порядку (в цикле генерируются числа от 1 до $n$, и для каждого из них производится операция, обратная ранжированию).

В конце главы обсуждается задача генерирования графов. Впрочем, алгоритмы на графах представлены более подробно в последующих главах *второй части* книги.

Среди книг по общим комбинаторным алгоритмам я порекомендую следующие:

- [Knu98] и [Knu11]. Первая книга представляет собой стандартное пособие по сортировке и поиску. А материал второй — генерирование перестановок, подмножеств, разбиений и деревьев — предположительно составляет содержимое первой части мифического Тома 4 работ Кнута.
- [Rus03] — Хотя Фрэнк Раски (Frank Ruskey) никогда официально не завершил ее, эта книга является стандартным пособием по генерированию комбинаторных объектов. Обзорную информацию о материале книге можно найти в Интернете, выполнив поиск по ключевым словам: `Ruskey Combinatorial Generation`.
- [KS99] — книга по комбинаторным алгоритмам. Кроме этого, в ней особое внимание уделяется алгебраическим задачам — таким как изоморфизм и симметрия;
- [PS03] — описание библиотеки Combinatorica, содержащей свыше 400 функций системы Mathematica для генерирования комбинаторных объектов и объектов теории графов (см. *разд. 22.1.8*). Авторы книги придерживаются особого взгляда на взаимодействие разных алгоритмов.

# 17.1. Сортировка

**Вход.** Множество из $n$ элементов.

**Задача.** Расположить элементы в возрастающем (или убывающем) порядке (рис. 17.1).

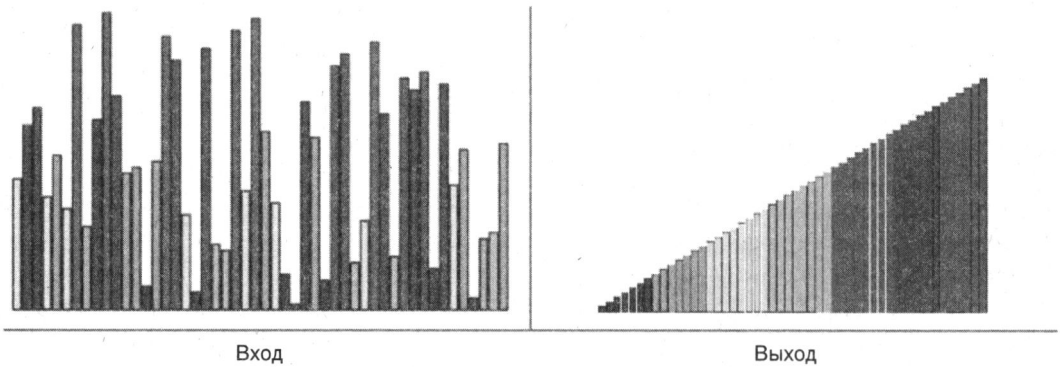



**Рис. 17.1.** Сортировка

### Обсуждение

Сортировка является фундаментальной алгоритмической задачей теории вычислительных систем. Для программиста освоение различных алгоритмов сортировки сродни изучению гамм для музыканта. Как показано в *разд. 4.2*, сортировка оказывается первым шагом в решении множества других алгоритмических задач, а правило «если не знаешь, как поступить, выполняй сортировку» — это одно из главных правил при разработке алгоритмов.

Сортировка также иллюстрирует большинство стандартных парадигм разработки алгоритмов. Программистам обычно известно так много разных алгоритмов сортировки, что зачастую им трудно принять решение, какой из них выбрать в конкретном случае. Следующие вопросы помогут вам в выборе алгоритма.

- *Сколько элементов нужно отсортировать?*

  Для небольших объемов данных ($n \leq 100$) не играет особой роли, какой из алгоритмов с квадратичным временем исполнения использовать. Сортировка вставками быстрее, проще и в меньшей степени чревата ошибками реализации, чем пузырьковая сортировка. Сортировка методом Шелла очень похожа на сортировку вставками, только намного быстрее. Однако для ее реализации необходимо знание правильной последовательности вставок, изложенной в книге [Knu98].

  Но если количество сортируемых элементов больше 100, то важно использовать алгоритм с временем исполнения $O(n \lg n)$, такой как пирамидальная сортировка, быстрая сортировка или сортировка слиянием. Разные программисты выбирают различные алгоритмы, руководствуясь личными предпочтениями. Поскольку трудно сказать, какой алгоритм быстрее, выбор того или иного алгоритма не имеет особого значения.

А когда количество элементов для сортировки превышает, скажем, 100 000 000, то пора начинать думать об использовании алгоритма сортировки данных вне памяти, который минимизирует обращения к диску или сетевому хранилищу. Алгоритмы обоих типов рассматриваются далее в этом разделе.

- *Содержатся ли в данных дубликаты?*

  Порядок сортировки полностью определен, если все элементы имеют разные ключи. Но когда какие-либо два элемента имеют одинаковый ключ, то для решения, какой элемент должен стоять первым, нужно применить дополнительный критерий. В случаях, когда это важно, очередность элементов с одинаковыми ключами определяется по вторичному ключу — например, по имени для одинаковых фамилий.

  А иногда очередность в отсортированном списке определяется по положению в исходном списке. Допустим, что 5-й и 27-й элементы исходного набора данных имеют одинаковый ключ. Тогда в отсортированном списке 5-й элемент должен находиться перед 27-м. То есть в случае совпадения ключей *стабильный* алгоритм сортировки сохраняет исходный порядок элементов в отсортированном списке. Большинство алгоритмов сортировки с квадратичным временем исполнения являются стабильными, в то время как многие из алгоритмов с временем исполнения $O(n \lg n)$ — нет. Если важна стабильность сортировки, то в функции сравнения, скорее всего, лучше использовать исходное положение элемента в качестве вторичного ключа, чем полагаться на реализацию алгоритма для обеспечения стабильности.

- *Что известно о данных?*

  Возможно, что для ускорения сортировки данных удастся воспользоваться их особенностями. Общая сортировка, имеющая время исполнения $O(n \lg n)$, — это операция быстрая, поэтому, если узким местом вашего приложения *действительно* является время, затрачиваемое на сортировку, можете считать, что вам повезло. Итак:

  - *данные уже частично отсортированы?*

    В таком случае определенные алгоритмы — например, алгоритм сортировки вставками, — работают быстрее, чем на полностью неотсортированных данных;

  - *что известно о распределении ключей?*

    Если ключи распределены произвольно или равномерно, то имеет смысл использовать корзинную сортировку или сортировку распределением. Просто помещаем ключи в корзины по их первой букве и выполняем рекурсию до тех пор, пока каждая корзина не достигнет размера, достаточно малого для сортировки полным перебором. Это очень эффективный подход, когда все ключи распределены в пространстве ключей равномерно. Но в случае сортировки списка группы однофамильцев производительность корзинной сортировки окажется весьма низкой;

  - *очень длинные или трудные для сравнения ключи?*

    При сортировке длинных текстовых строк, возможно, имеет смысл выполнить предварительную сортировку по сравнительно короткому префиксу (скажем, длиной в 10 символов), после чего произвести сортировку по полному ключу.

Этот метод особенно подходит для сортировки на внешних носителях (см. далее), при которой нежелательно расходовать быструю память на хранение несущественных деталей.

Можно также прибегнуть к поразрядной сортировке, имеющей линейное время исполнения относительно количества символов в файле, что гораздо лучше времени $O(n \lg n)$, которое еще должно быть помножено на стоимость сравнения двух ключей;

- *невелик диапазон возможных ключей?*

  Если требуется отсортировать подмножество из, скажем, $n/2$ разных целых чисел, значение каждого из которых лежит в диапазоне от 1 до $n$, то самым быстрым алгоритмом будет создать $n$-элементный битовый вектор, установить соответствующие ключам биты, а потом отсканировать вектор слева направо и доложить о позициях с установленными битами.

♦ *Необходимо ли обращаться к диску?*

При сортировке больших объемов информации все сортируемые данные могут не помещаться в память. В таких случаях мы имеем дело с задачей *внешней сортировки* (external sorting), для которой требуется применение внешнего устройства хранения данных. Первоначально в качестве внешних устройств хранения данных использовались накопители на магнитной ленте, и в книге [Knu98] описываются разнообразные сложные алгоритмы для эффективного слияния данных с разных накопителей. В настоящее время задействуются виртуальная память и обмен данными с диском. С виртуальной памятью сможет работать любой алгоритм сортировки, но некачественные алгоритмы будут тратить значительное время на перемещение данных между памятью и диском.

Самый простой подход к внешней сортировке — это загрузка данных в B-дерево с последующим симметричным его обходом, позволяющим читать ключи в отсортированном порядке. Однако наиболее высокопроизводительные программы сортировки основаны на сортировке многоканальным слиянием. При этом данные разбиваются на несколько файлов, и каждый из них сортируется с применением быстрой внутренней сортировки, после чего отсортированные файлы сливаются поэтапно с использованием двух- или $k$-канального слияния. Производительность можно оптимизировать, применив сложные схемы слияния и управления буфером, учитывающие свойства внешнего устройства хранения.

Самым лучшим общим алгоритмом внутренней сортировки является быстрая сортировка (см. *разд. 4.2*), хотя для получения максимальной производительности придется повозиться с ее настройкой. В действительности, намного лучше использовать для этого алгоритма библиотечную функцию вашего компилятора, чем пытаться разработать свою реализацию. Неудачная реализация быстрой сортировки будет работать медленнее, чем плохая реализация пирамидальной сортировки.

Если же вы настроены на создание своей реализации алгоритма быстрой сортировки, то используйте следующие эвристические методы, которые на практике оказывают большое влияние на производительность:

- *применяйте рандомизацию.*

  Выполнив произвольную перестановку ключей перед сортировкой (см. *разд. 17.4*), вы, возможно, добьетесь того, что время сортировки почти отсортированных данных не будет квадратичным;

- *выбирайте средний элемент из трех.*

  Для выбора опорного элемента используйте средний по величине элемент из первого, последнего и центрального элемента массива, чтобы повысить вероятность разбиения массива на приблизительно одинаковые части. По результатам экспериментов для больших подмассивов следует составлять выборку из большего количества элементов, а для небольших — из меньшего;

- *для сортировки вставками выбирайте подмассивы небольшого размера.*

  Переключение с рекурсивной быстрой сортировки на сортировку вставками имеет смысл только при небольшом размере подмассивов — например, не превышающем 20 элементов. Оптимальный размер подмассивов для такой реализации определяется экспериментальным путем;

- *обрабатывайте наименьшую* порцию данных *в первую очередь.*

  Объем требуемой памяти можно минимизировать, сначала обрабатывая меньшие порции данных. Так как каждый следующий рекурсивный вызов занимает в стеке память, превышающую память, отведенную под предыдущий вызов, максимум в полтора раза, то потребуется только $O(\lg n)$ стековой памяти.

И в любом случае, прежде чем приступать к реализации алгоритма быстрой сортировки, прочитайте статью [Ben92b].

### Реализации

Можно предположить, что наилучшей программой сортировки с открытым кодом является GNU sort, представляющая собой часть библиотеки основных утилит GNU. Дополнительную информацию см. по адресу **http://www.gnu.org/software/coreutils/**. Существуют и коммерческие поставщики высокопроизводительных программ внешней сортировки — такие как Cosort (**www.iri.com**), Syncsort (**www.syncsort.com**) и Ordinal Technology (**www.ordinal.com**).

Современные языки программирования содержат библиотеки эффективных реализаций сортировки, поэтому вам вряд ли когда-нибудь потребуется разрабатывать для этого собственную процедуру. Стандартная библиотека C содержит функцию `qsort`, которая, на мой взгляд, и представляет собой обобщенную реализацию быстрой сортировки. Библиотека STL языка C++ предоставляет методы `sort` и `stable_sort`. Более подробное руководство по использованию библиотеки STL и стандартной библиотеки C++ можно найти в книгах [Jos12] и [Mey01]. Стандартный пакет утилит Java SE java.util содержит небольшую библиотеку структур данных Java Collections (JC), предоставляющую, в частности, классы `SortedMap` и `SortedSet`.

Высокопроизводительные системы сортировки распределяют работу по нескольким машинам. Параллельную корзинную сортировку можно сравнительно легко реализовать с помощью систем MapReduce — например, Hadoop (см. [Whi12]). Об этом чем-

пионе по сортировке терабайта данных сообщается в [O08]. Эффективные алгоритмы сортировки для графических процессоров рассматриваются в [SHG09].

Существует множество веб-сайтов, содержащих анимационные апплеты для всех фундаментальных алгоритмов сортировки, и за работой многих из них довольно интересно наблюдать. Безусловно, сортировка является классической задачей для анимации алгоритмов. В Интернете можно найти множество образцов подобных анимаций, выполнив поиск по таким, например, ключевым словам: `sorting animation` или `визуализация алгоритмов сортировки`.

### *Примечания*

Самая лучшая книга по сортировке, которая когда-либо была написана, — это книга [Knu98]. Ей почти пятьдесят лет, но читать ее по-прежнему интересно. Одной из областей сортировки, разработанной после первого издания этой книги, является сортировка предварительно отсортированных данных, исследуемая в работе [ECW92].

В Timsort — популярном кандидате в наиболее быстрые на практике алгоритмы сортировки — используются естественно упорядоченные последовательности, дающие возможность получить время исполнения, равное $O(n \log p)$, где $p$ обозначает количество отсортированных последовательностей, обнаруженных во входных данных. Подробности см. в [AJNP18].

Описания основных алгоритмов внутренней сортировки содержатся в каждом учебнике по алгоритмам. Алгоритм пирамидальной сортировки разработан в 1964 году Дж. Вильямсом (J. Williams) — см. [Wil64]. Алгоритм быстрой сортировки разработан в 1960 году Чарльзом Хоаром (Charles Hoare) — см. [Hoa62], а его всесторонний анализ и реализация осуществлены Р. Седжвиком (R. Sedgewick) — см. [Sed78]. Первая реализация алгоритма сортировки слиянием (на компьютере EDVAC в 1945 году) была выполнена фон Нейманом. Полное изложение истории сортировки, берущей начало во времена электронно-вычислительных машин, работавших на перфокартах, вы найдете в книге [Knu98].

Джим Грей (Jim Gray) инициировал проведение ежегодных соревнований по высокопроизводительной сортировке, и на веб-сайте **http://sortbenchmark.org/** публикуется информация о текущих и предыдущих их результатах, которые могут как вдохновлять, так и расстраивать — в зависимости от вашей точки зрения. Достигнутый прогресс вдохновляет (тестовые экземпляры размером в миллион записей, которые использовались в первых соревнованиях, теперь кажутся незначительными), но лично меня расстраивает тот факт, что вопросы управления системой и памятью оказываются гораздо важнее комбинаторно-алгоритмических аспектов сортировки. В настоящее время создание высокопроизводительных программ сортировки включает в себя работу над алгоритмами как требовательными к кэшированию (см. [LL99]), так и нечувствительными к нему (см. [BFV07]).

Для сортировки существует хорошо известная нижняя граница $\Omega(n \lg n)$ на модели алгебраического дерева решений (см. [BO83]). Задача определения точного количества сравнений, необходимого для сортировки $n$ элементов при небольших значениях $n$, вызвала значительный интерес у исследователей. Описание этой задачи можно найти в [Aig88] и [Raw92], а последние результаты — в [Pec04] и [Pec07].

Но такая нижняя граница не соблюдается при других моделях вычислений. В работе [FW93] представляется алгоритм сортировки со сложностью $O(n\sqrt{\lg n})$ по модели вычислений, позволяющей выполнять арифметические операции над ключами. Обзор алгоритмов быстрой сортировки по таким нестандартным моделям вычислений приведен в [And05].

### Родственные задачи

Словари (см. *разд. 15.1*), поиск (см. *разд. 17.2*), топологическая сортировка (см. *разд. 18.2*).

## 17.2. Поиск

**Вход.** Набор $S$ из $n$ ключей и ключ запроса $q$.

**Задача.** Определить местоположение ключа $q$ в наборе ключей $S$ (рис. 17.2).

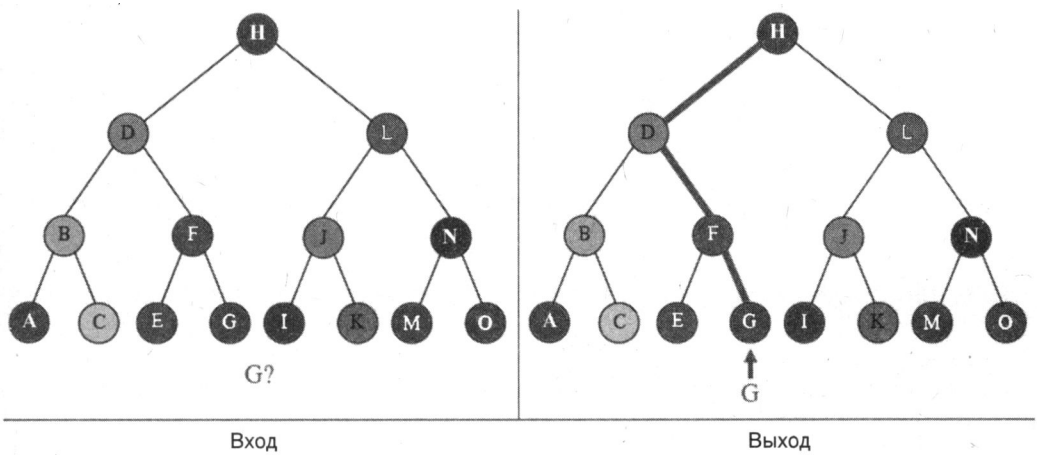



**Рис. 17.2.** Поиск в наборе ключей

### Обсуждение

Для разных людей слово «поиск» означает разные понятия. Так, поиск глобального максимума или минимума функции является задачей *безусловной оптимизации*, которая рассматривается в *разд. 16.5*. Программы для игры в шахматы выбирают свой лучший ход, выполняя исчерпывающий перебор/поиск всех возможных ходов, используя вариант поиска с возвратом (см. *разд. 9.1*).

Мы же ставим перед собой задачу найти ключ в списке, массиве или дереве. Эффективный доступ к наборам ключей для вставки и удаления предоставляют нам словарные структуры данных (см. *разд. 15.1*). Типичными представителями таких структур являются деревья двоичного поиска и хеш-таблицы.

Задачу поиска мы рассматриваем без привязки к словарям, поскольку в нашем распоряжении имеются более простые и эффективные решения для статического поиска без вставок и удалений. При правильном использовании во внутреннем цикле такие простые структуры данных, как деревья двоичного поиска и хеш-таблицы, могут обеспечить значительное повышение производительности. Кроме того, идеи двоичного поиска и самоорганизации применимы при решении других задач, что оправдывает наше внимание к ним.

Мы рассмотрим два основных метода поиска: последовательный поиск и двоичный поиск. Оба достаточно просты, но у них имеются интересные варианты. При *последо-*

*вательном поиске* мы движемся по списку ключей с самого начала и сравниваем каждый последующий элемент с ключом поиска $q$, пока не найдем совпадающий или не достигнем конца списка. При *двоичном поиске* работа производится с отсортированным списком или массивом ключей. Чтобы найти ключ $q$, мы сравниваем значение $q$ со значением среднего ключа массива $S_{n/2}$. Если значение ключа $q$ меньше, чем значение ключа $S_{n/2}$, то искомый ключ должен находиться в верхней половине массива $S$, в противном случае он будет расположен в его нижней половине. Повторяя этот процесс на той части массива, которая должна содержать элемент $q$, мы находим его за $[\lg n]$ сравнений, что намного лучше, чем ожидаемые $n/2$ сравнений при последовательном поиске. Дополнительную информацию по двоичному поиску см. в *разд. 5.1*.

Последовательный поиск — это простейший алгоритм, и, скорее всего, он будет самым быстрым, когда количество элементов не превышает 20. При ста и более элементах двоичный поиск окажется более эффективным, чем последовательный, и оправдает затраты на сортировку при большом количестве запросов. Но для принятия решения о выборе варианта алгоритма необходимо учитывать и следующие факторы:

- *сколько времени отводится на программирование?*

  Известно, что алгоритм двоичного поиска сложен для программирования. Между изобретением алгоритма и опубликованием первой *правильной* его реализации прошло целых семнадцать лет! Поэтому я советую вам начать с одной из реализаций, описанных далее. Полностью протестируйте ее, создав программу, которая выполняет поиск каждого ключа в наборе $S$, а также промежуточных значений ключей;

- *какова частота обращений к разным ключам?*

  Определенные английские слова (например, «the») встречаются намного чаще, чем другие (например, «defenestrate»). Количество сравнений в последовательном поиске можно уменьшить, поместив часто употребляемые слова вверху списка, а более редкие — внизу. Неравномерность запросов является скорее правилом, чем исключением. Слова во многих языках распределяются согласно *степенным законам*. Классическим примером является распределение слов в английском языке, которое довольно точно моделируется *законом Ципфа*. Согласно этому закону $i$-й по частоте обращения ключ выбирается с вероятностью, которая в $(i-1)/i$ раз превышает вероятность выбора ключа $(i-1)$-го по популярности для всех $1 \le i \le n$.

  Знание частоты обращений легко использовать в последовательном поиске. Но в двоичном дереве дело обстоит сложнее. Мы хотим, чтобы часто употребляемые ключи располагались возле корня (так мы быстрее найдем их), но не за счет разбалансирования дерева и превращения двоичного поиска в последовательный. Решение этой проблемы заключается в применении алгоритма динамического программирования для построения *оптимального дерева* двоичного поиска. Критически важным здесь является то обстоятельство, что каждый возможный корневой узел $i$ делит пространство ключей на две части (слева и справа от $i$), и каждую из них можно представить оптимальным деревом двоичного поиска для меньшего поддиапазона ключей. Корень оптимального дерева выбирается таким образом, чтобы минимизировать ожидаемую стоимость поиска в получившемся разбиении;

- *может ли частота обращений измениться со временем?*

  Для предварительной сортировки списка или дерева с целью воспользоваться асимметричной закономерностью доступа требуется знать эту закономерность доступа заранее. Для многих приложений получение такой информации может оказаться трудной задачей. Лучший подход — использовать *самоорганизующиеся списки*, в которых порядок ключей изменяется в ответ на запросы. При этом самой лучшей схемой самоорганизации является перемещение ключа, к которому выполнялось самое последнее обращение, в начало списка. Таким образом ключи с наибольшим числом обращений продвигаются в начало списка, а с наименьшим — к концу. Отслеживать частоту обращений нет необходимости — мы просто перемещаем ключ при обращении к нему. В самоорганизующихся списках также работает принцип временной локальности, поскольку существует вероятность повторных обращений к любому ключу. Таким образом, пользующийся спросом ключ будет удерживаться вверху списка на протяжении всей последовательности обращений к нему, даже если в прошлом другие ключи пользовались бóльшим спросом.

  Принцип самоорганизации может расширить размер полезного диапазона последовательного поиска, однако когда количество ключей превышает 100, то следует переключаться на двоичный поиск. Но также можно рассмотреть возможность использования *косых деревьев*, являющихся деревьями двоичного поиска, в которых элемент, к которому было выполнено обращение, перемещается в корневой узел. Эти деревья дают гарантию отличной амортизированной производительности;

- *известно ли примерное местоположение ключа?*

  Допустим, мы знаем, что требуемый ключ находится справа от позиции $p$, причем на небольшом расстоянии. Если мы правы в нашем предположении, то последовательный поиск быстро возвратит нужный элемент, однако за ошибку нам придется дорого заплатить. Гораздо лучше проверять ключи через возрастающие интервалы справа от $p$ ($p + 1$, $p + 2$, $p + 4$, $p + 8$, $p + 16$, ...), пока мы не дойдем до ключа, расположенного справа от целевого. Таким способом мы определяем окно, содержащее целевой элемент, и теперь мы сможем найти его, применив обычный двоичный поиск.

  Этот *односторонний двоичный поиск* возвращает целевой ключ в позиции $p + l$ за самое большее $2\lceil \lg l \rceil$ сравнений, т. е. он работает быстрее двоичного поиска, если $l \ll n$, и такая производительность никогда не станет намного хуже. Односторонний двоичный поиск особенно полезен при неограниченном поиске — например, при извлечении корня числа;

- *находится ли структура данных на внешнем устройстве?*

  Когда количество ключей слишком велико, двоичный поиск теряет свой статус лучшего метода поиска. Он будет метаться по пространству ключей в поисках средней точки для сравнения с целевым ключом, и для каждого из этих сравнений потребуется загрузить новую страницу из внешнего устройства. Намного лучше использовать такие структуры данных, как B-деревья (см. *разд. 15.1*) или деревья ван Эмде Боаса (см. *подраздел «Примечания»* далее), которые собирают ключи в страницы, чтобы свести к минимуму количество обращений к диску в каждом поиске;

- *можно ли угадать местоположение ключа?*

  В *интерполяционном поиске* мы используем свое знание распределения ключей, чтобы угадать следующую позицию для сравнения. Кстати, при поиске в телефонной книге более уместно было бы говорить об интерполяционном, а не о двоичном поиске. Допустим, что мы ищем номер телефона человека по имени `Washington, George`. При этом можно уверенно делать первое сравнение где-то в третьей четверти книги, фактически выполняя два сравнения по цене одного.

  Хотя идея интерполяционного поиска кажется привлекательной, я рекомендую не использовать его по трем причинам. Во-первых, придется выполнить большую работу, чтобы оптимизировать алгоритм поиска, прежде чем можно будет надеяться получить лучшую производительность, чем при двоичном поиске. Во-вторых, даже если вы и повысите производительность по сравнению с двоичным поиском, то вряд ли настолько, чтобы оправдать вложенные усилия. И наконец, в-третьих, ваша программа будет намного менее надежна и эффективна при смене локали — например, при работе с французским текстом вместо английского.

### Реализации

Элементарные алгоритмы последовательного и двоичного поиска настолько просты, что следует рассматривать возможность их самостоятельной реализации. Тем не менее стандартная библиотека C содержит процедуру `bsearch`, которая, на мой взгляд, и представляет собой обобщенную реализацию двоичного поиска. Библиотека STL языка C++ предоставляет итераторы `find` (последовательный поиск) и `binary_search` (двоичный поиск). Стандартный пакет утилит Java `java.util` предоставляет процедуру `binarySearch`.

Многочисленные реализации приводятся во многих учебниках по структурам данных. Реализации косых деревьев и других поисковых структур на языках C++ и Java даются в книгах [SW11] (**https://algs4.cs.princeton.edu/code/**)) и [Wei11] (**http://www.cs.fiu.edu/~weiss/**)).

#### Примечания

В книге ([MS18]) приводится обзор современного состояния дел в области словарных структур данных. Другие обзоры представлены в книгах [MT90b] и [GBY91]. Книга [Knu97a] содержит детальный анализ и описание основных алгоритмов поиска и словарных структур данных, однако в ней отсутствуют такие современные структуры данных, как красно-черные и косые деревья.

Очередная позиция сравнения при линейном интерполяционном поиске в массиве отсортированных чисел определяется по следующей формуле:

$$next = (low - 1) + \left\lceil \frac{q - S[low - 1]}{S[high + 1] - S[low - 1]} \times (high - low + 1) \right\rceil,$$

где $q$ — числовой ключ запроса, а $S$ — отсортированный массив числовых значений. Если ключи распределены равномерно и выбираются независимым образом, то ожидаемое время поиска равно $O(\lg \lg n)$ (см. [DJP04] и [PIA78]). На практике же такого распределения ключей ожидать не стоит.

Неравномерность распределения обращений можно использовать в деревьях двоичного поиска, организовав их таким образом, чтобы часто запрашиваемые ключи находились

возле корня, что позволит минимизировать время поиска. Такие оптимальные деревья поиска можно создавать за время $O(n \lg n)$ посредством динамического программирования (см. [Knu98]). В книге [SW86] описан нетривиальный алгоритм для эффективного преобразования двоичного дерева в дерево минимальной высоты (оптимально сбалансированное) посредством операций ротации.

Метод ван Эмде Боаса для создания двоичного дерева (или отсортированного массива) обеспечивает более высокую производительность при работе с внешними устройствами хранения данных, чем двоичный поиск, но за счет более сложной реализации. Информацию об этой и других нечувствительных к кэшированию структурах см. в [ABF05].

На квантовых компьютерах алгоритм Гровера позволяет выполнять поиск в неупорядоченной базе данных за время $O(\sqrt{n})$ (подробности см. в [NC02]). На базовом интуитивном уровне понятно, что квантовые компьютеры работают с вероятностями, изначально одинаковыми для $n$ элементов в суперпозиции. Но простая операция усиления амплитуды может повысить вероятность целевого элемента в $\sqrt{n}$ раз. После $O(\sqrt{n})$ таких усилений выборка целевого элемента становится высоковероятной.

### Родственные задачи

Словари (см. *разд. 15.1*), сортировка (см. *разд. 17.1*).

## 17.3. Поиск медианы и выбор элементов

**Вход.** Набор из $n$ чисел (или ключей), целое число $k$.

**Задача.** Найти ключ такой, что $k$ ключей из $n$ меньше или равны ему (рис. 17.3).

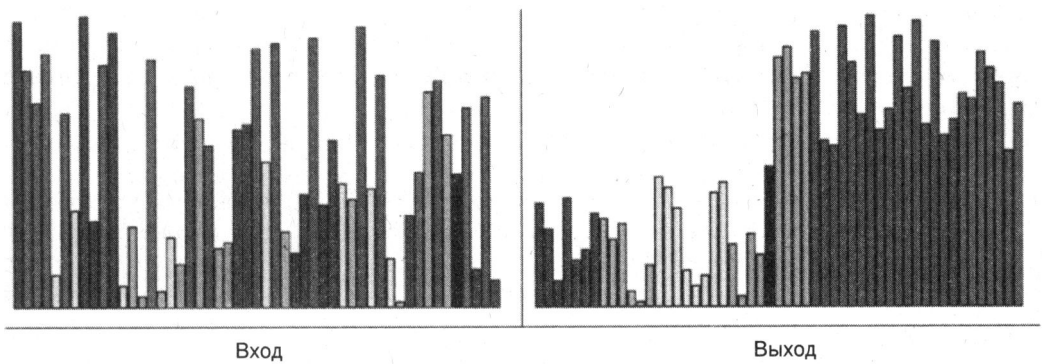



**Рис. 17.3.** Сортировка по среднему значению

### Обсуждение

Поиск медианы (элемента, среднего по местоположению) является важной задачей в статистике, поскольку он возвращает более устойчивое понятие, чем *среднее значение*. На среднее значение доходов всех людей, опубликовавших научные работы по сортировке, оказывает сильное влияние факт присутствия среди них Билла Гейтса (см. [GP79]), но его влияние на *медиану* доходов сводится к нейтрализации одного малообеспеченного аспиранта.

Задача поиска медианы представляет собой частный случай общей *задачи о выборе*, в которой требуется выбрать $i$-й элемент в отсортированном списке. Задача выбора возникает во многих приложениях, примеры которых приводятся в следующем списке:

- *фильтрация элементов с резко отклоняющимися значениями.*

  При работе с зашумленными данными часто бывает полезно отбросить 10% самых больших и самых меньших значений. Для этого выбираются элементы, соответствующие 10-му и 90-му процентилям, после чего значения, не попадающие в диапазон, ограниченный этими точками, отфильтровываются после сравнения каждого элемента с выбранными границами;

- *идентификация наиболее перспективных кандидатов.*

  Компьютерная программа для игры в шахматы может быстро оценить все возможные следующие ходы и затем рассмотреть 25% самых лучших ходов более пристально. Здесь опять выбираются границы, по которым затем отфильтровываются заведомо бесперспективные ходы;

- *применение децилей и аналогичные способы деления диапазона.*

  Удобным способом представления распределения доходов населения является отображение зарплат людей на графике по равномерным интервалам, границы которых устанавливаются, например, по 10-му перцентилю, по 20-му перцентилю, и т. д. Вычисление этих значений просто сводится к выбору элементов из соответствующего интервала;

- *статистика порядков.*

  Особенно интересные частные случаи задачи выбора — это поиск наименьшего элемента ($k = 1$), наибольшего элемента ($k = n$) и медианного элемента ($k = n/2$).

Среднее значение $n$ чисел можно вычислить за линейное время, сложив все элементы и разделив получившуюся сумму на $n$. Но поиск медианы представляет более трудную задачу. Алгоритмы вычисления медианы можно легко обобщить к произвольной выборке. Приведу вопросы, на которые следует ответить при поиске медианы и выборе элементов.

- *Какой должна быть скорость работы?*

  Самый простой алгоритм вычисления медианы сортирует элементы за время $O(n \lg n)$, после чего возвращает элемент, занимающий $(n/2)$-ю позицию. Дополнительный плюс этого алгоритма состоит в том, что он выдает намного больше информации, чем одна лишь медиана, позволяя выбрать любой $k$-й элемент (для $1 \le k \le n$) за постоянное время после сортировки. Но если вам требуется только медиана, то для ее поиска существуют более быстрые алгоритмы.

  В частности, одним из таких алгоритмов является алгоритм QuickSelect на основе быстрой сортировки с *ожидаемым временем* исполнения $O(n)$. Этот алгоритм работает следующим образом. Выбираем в наборе данных произвольный элемент и используем его для разбиения данных на два набора: один из них содержит элементы меньшие, чем элемент-разделитель, а другой — большие. Зная размер этих наборов, мы знаем позицию элемента-разделителя в исходном наборе, а следовательно,

и расположение медианы — слева или справа от разделителя. Рекурсивно выполняем эту процедуру на соответствующем наборе данных, пока не найдем медиану. Нам понадобится в среднем $O(\lg n)$ итераций, а затраты на выполнение каждой последующей итерации примерно равны половине стоимости предыдущей. Времена выполнения каждой итерации образуют геометрическую прогрессию, которая сходится к линейному времени исполнения. Впрочем, если нам очень не повезет, то время исполнения может оказаться таким же, как и для наихудшего случая быстрой сортировки, — т. е. $\Theta(n^2)$.

Более сложные алгоритмы находят медиану за линейное время в наихудшем случае. Однако для практического применения лучше всего подойдет алгоритм с ожидаемым линейным временем исполнения.

- *Как поступить, если мы видим каждый элемент только один раз?*

На больших наборах данных операции выбора элементов и вычисления медианы становятся слишком дорогими, поскольку для них требуется выполнить несколько проходов по данным, хранящимся на внешних устройствах. А у приложений, работающих с потоковыми данными, объем данных слишком велик для того, чтобы их сохранять. В результате их повторное рассмотрение (и, следовательно, вычисление точного значения медианы) невозможно. В таких случаях лучше построить менее объемную модель данных для дальнейшего анализа — например, на основе децилей моментов распределения (где $k$-й момент потока $x$ определяется как $F_k = \sum_i x_i^k$).

Одним из решений такой задачи может стать произвольная выборка. Решение, сохранять ли значение, принимается на основе броска монеты, у которой вероятность выпадения стороны, соответствующей сохранению, достаточно низка, чтобы не переполнить буфер. Скорее всего, медиана сохраненных выборок будет близкой к медиане исходного набора данных. В качестве альтернативного варианта можно выделить определенную область памяти для хранения, например, значений децилей больших блоков, после чего объединить децили, чтобы получить более точные границы.

- *Насколько быстро можно найти моду?*

Кроме среднего и медианы существует еще одно понятие средней величины — *мода*. Мода определяется как наиболее часто встречающийся элемент набора данных. Наилучший алгоритм для вычисления моды сортирует набор данных за время $O(n \log n)$, в результате чего создаются последовательности одинаковых элементов. Перебирая отсортированные данные слева направо, мы можем найти длину самой протяженной последовательности одинаковых элементов и вычислить моду за общее время $O(n \log n)$.

Моду можно также вычислить за *ожидаемое* линейное время, используя хеширование, но такой алгоритм для *наихудшего случая* невозможен. Задача проверки наличия двух идентичных элементов в наборе имеет нижнюю временную границу $\Omega(n \log n)$. Задача поиска одинаковых элементов эквивалентна задаче поиска нескольких мод. Существует возможность, по крайней мере теоретическая, улучшить время исполнения для больших значений моды за счет использования быстрых вычислений медианы.

### Реализации

Библиотека STL на языке C++ содержит универсальный метод выбора элемента (`nth_element`), реализованный на основе алгоритма с ожидаемым линейным временем исполнения. Библиотека STL и стандартная библиотека C++ подробно описаны в книгах [Jos12], [Mey01] и [MDS01].

#### Примечания

Алгоритм с ожидаемым линейным временем исполнения для вычисления медианы и выбора элемента разработан Хоаром (Hoare) — см. [Hoa61]. В работе [FR75] представлен алгоритм, выполняющий в среднем меньшее число сравнений. Хорошие описания алгоритма выбора с линейным временем исполнения приведены в книгах [BvG99], [CLRS09] и [Raw92], среди которых [Raw92] является наиболее информативной.

Потоковые алгоритмы широко применяются для работы с большими наборами данных. Подробный обзор этих алгоритмов можно найти в книгах [Mut05] и [CH09].

Значительный теоретический интерес представляет задача определения *точного* количества сравнений, достаточных для вычисления медианы $n$ элементов. Алгоритм с линейным временем исполнения, описываемый в работе [BFP+72], доказывает, что достаточно $c \cdot n$ сравнений, но мы хотим знать, чему равно $c$. В работе [DZ99] доказано, что для вычисления медианы достаточно выполнить $2{,}95n$ сравнений. Эти алгоритмы стремятся минимизировать количество сравнений элементов, но не общее количество операций, и поэтому на практике не оказываются намного быстрее существующих. Кроме того, они оставляют на месте наилучшую в настоящий момент нижнюю границу времени вычисления медианы, равную $(2 + \varepsilon)$ сравнениям.

Исследование пределов комбинаторных алгоритмов для решения задач выбора представлено в книге [Aig88]. Оптимальный алгоритм для вычисления моды приведен в работе [DM80].

### Родственные задачи

Очереди с приоритетами (см. *разд. 15.2*), сортировка (см. *разд. 17.1*).

## 17.4. Генерирование перестановок

**Вход.** Целое число $n$.

**Задача.** Создать: все возможные перестановки, или случайную перестановку, или очередную перестановку размером $n$ (рис. 14.4).

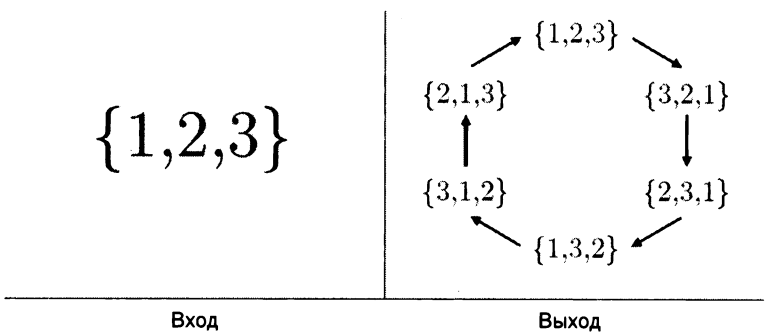



**Рис. 17.4.** Перестановки

### Обсуждение

*Перестановкой* называется упорядоченный набор элементов. Во многих алгоритмических задачах требуется найти наилучший способ упорядочения набора объектов. В качестве примеров можно привести *задачу коммивояжера* (определение порядка посещения $n$ городов, имеющего наименьшую стоимость), *задачу уменьшения ширины ленты* (упорядочивание вершин графа в линию таким образом, чтобы минимизировать длину самого длинного ребра) и *задачу изоморфизма графа* (упорядочивание вершины графа таким образом, чтобы он был идентичным другому графу). Любой алгоритм для предоставления точного решения таких задач должен создавать в процессе решения последовательность перестановок.

Из $n$ элементов можно создать $n!$ перестановок. С увеличением $n$ количество перестановок возрастает так быстро, что не стоит надеяться сгенерировать все перестановки даже для $n \geq 15$, поскольку 15! = 1 307 674 368 000. Подобные числа должны охладить пыл любого человека, пытающегося решить задачу методом исчерпывающего перебора, и помочь ему понять важность генерирования случайных перестановок.

При обсуждении генерирования перестановок основным является понятие *порядка* — т. е. последовательности, в которой они создаются. Наиболее естественным порядком генерирования перестановок является *лексикографический* — такой, в котором они находились бы после сортировки. Например, лексикографический порядок перестановок для $n = 3$ таков: {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}. Хотя лексикографический порядок эстетически весьма привлекателен, часто он не дает никаких определенных преимуществ. Например, при поиске в коллекции файлов порядок, в котором рассматриваются имена файлов, не имеет значения при условии, что в конечном итоге вы просмотрите их все. Более того, если лексикографический порядок не требуется, то нередко получаются более быстрые и простые алгоритмы.

Для создания перестановок существуют две разные парадигмы: метод ранжирования/деранжирования и метод инкрементального изменения. Второй метод более эффективен, но первый применим для решения более широкого класса задач. Ключевым моментом метода ранжирования/деранжирования является определение функций `Rank` и `Unrank` на всех перестановках $p$ и целых числах $n$, $m$, где $|p| = n$ и $0 \leq m \leq n! - 1$.

- Функция ранжирования `Rank`($p$) определяет позицию перестановки $p = \{p_1, ..., p_n\}$ в заданном порядке генерирования. Типичная функция ранжирования является рекурсивной. Например, при базовом выражении `Rank`({1}) = 0 общая формула выглядит так:

$$\texttt{Rank}(p) = (p_1 - 1)\cdot(|p| - 1)! + \texttt{Rank}(p_2, ..., p_{|p|}).$$

  Поскольку предполагается, что любая перестановка $p$ представляет собой набор различных целых чисел от 1 по |p|, то чтобы функция оставалась правильной, необходимо присвоить новые метки элементам меньшей перестановки, чтобы отображать удаленный первый элемент. Этим объясняется, почему {1, 3} волшебным образом превращается в {1, 2}, как показано здесь:

$$\texttt{Rank}(\{2, 1, 3\}) = (2-1) \cdot 2! + \texttt{Rank}(\{1, 2\}) = 2 + (1-1) \cdot 1! + \texttt{Rank}(\{1\}) = 2.$$

- Функция деранжирования `Unrank`($m$, $n$) возвращает перестановку в позиции $m$ из $n!$ перестановок $n$ элементов. Типичная функция деранжирования рекурсивно опре-

деляет, сколько раз $(n-1)!$ встречается в $m$. Выражение `Unrank`(2,3) говорит нам, что первый элемент перестановки должен быть «2», поскольку $(2-1)\cdot(3-1)! \le 2$, но $(3-1)\cdot(3-1)! > 2$. Удаление $(2-1)\cdot(3-1)!$ из $m$ оставляет меньшую задачу `Unrank`(0,2). Ранг 0 соответствует полному упорядочению. Полным упорядочением на двух оставшихся элементах (поскольку 2 уже было использовано) является {1, 3}, поэтому `Unrank`(2, 3) = {2, 1, 3}.

Что именно делают функции `Rank` и `Unrank`, не имеет такого большого значения, как то обстоятельство, что они должны быть обратными друг другу. Иными словами, $p$ = `Unrank`(`Rank`($p$), $n$) для всех перестановок $p$. Определив функции ранжирования и деранжирования для перестановок, мы можем решать многие родственные задачи. В частности:

- *генерирование следующей перестановки.*

  Чтобы определить следующую по порядку после $p$ перестановку, мы можем выполнить `Rank`($p$), добавить к результату 1, после чего выполнить `Unrank`($p$). Аналогичным образом вычисляется и перестановка перед $p$: `Unrank`(`Rank`($p$) – 1,|$p$|). Выполнение функции `Unrank` над последовательностью целых чисел от 0 до $n!-1$ будет равнозначно генерированию всех перестановок;

- *генерирование случайных перестановок.*

  Выбрав случайное целое число от 0 до $n!-1$, а потом вызвав для него функцию `Unrank`, мы получим действительно случайную перестановку;

- *отслеживание набора перестановок.*

  Допустим, мы хотим генерировать случайные перестановки, но предпринимать некоторое действие только в том случае, когда получаем новую перестановку. Для отслеживания уже созданных перестановок можно создать битовый вектор (см. *разд. 15.5*) из $n!$ битов и устанавливать в нем бит $i$ при повторном генерировании перестановки $p$=`Unrank`($i$, $n$). Подобный метод использовался с подмножествами из $k$ элементов в приложении для игры в лото, описанном в *разд. 1.8*.

Метод ранжирования/деранжирования лучше всего подходит для небольших значений $n$, поскольку вычисление $n!$ быстро приведет к переполнению. Методы инкрементальных изменений включают в себя определение операций `next` и `previous` для преобразования одной перестановки в другую — обычно путем обмена местами двух элементов. Здесь сложным моментом является планирование таких обменов местами, чтобы перестановки не повторялись, пока не будут сгенерированы все $n!$ перестановки. Для примера на рис. 17.4 показано упорядочение шести перестановок множества {1, 2, 3} с использованием одного обмена местами между соседними перестановками.

Алгоритмы, созданные на основе методов инкрементальных изменений для генерирования последовательности перестановок, достаточно сложны, но настолько лаконичны, что их можно реализовать программой, состоящей из десятка строк (ссылки на существующий код приведены далее — в *подразделе «Реализации»*). Так как при инкрементальном изменении выполняется только один обмен, то эти алгоритмы могут быть чрезвычайно быстрыми, — в среднем с постоянным временем исполнения, не зависящим от размера перестановки! Секрет заключается в использовании $n$-элементного

массива для представления перестановки, чтобы облегчить обмен местами. В некоторых приложениях важно только различие между перестановками. Например, в программе поиска оптимального маршрута коммивояжера методом исчерпывающего перебора стоимостью маршрута, связанного с новой перестановкой, станет стоимость предшествующей перестановки с добавлением или удалением четырех ребер.

До сих пор мы предполагали, что все элементы перестановок отличаются друг от друга. Однако если их множество содержит дубликаты (иными словами, является *мультимножеством*), то можно сэкономить значительное время и усилия, не рассматривая одинаковые перестановки. Например, для множества {1, 1, 2, 2, 2} имеется не 120 разных перестановок, а только десять. Чтобы избежать повторений, перестановки нужно генерировать в лексикографическом порядке методом поиска с возвратом.

Генерирование случайных перестановок представляет собой несложную, но важную задачу, с которой разработчикам приходится часто сталкиваться и с которой они не всегда справляются. Для правильного решения этой задачи нужно использовать *алгоритм тасования Фишера — Йейтса* (Fisher — Yates shuffle), состоящий из двух строчек кода с линейным временем исполнения. Предполагается, что функция `Random`[*i*, *n*] генерирует случайное целое число в диапазоне между *i* и *n* включительно (листинг 17.1).

**Листинг 17.1. Алгоритм тасования Фишера — Йейтса**

```
for i = 1 to n do a[i] = i;
for i = 1 to (n - 1) do swap[a[i], a[Random[i, n]]];
```

Далеко не очевидно, что этот алгоритм генерирует все перестановки случайным образом с равномерным распределением. Если вы так не думаете, предоставьте убедительное объяснение, почему следующий алгоритм не генерирует равномерно распределенные перестановки (листинг 17.2).

**Листинг 17.2. Алгоритм тасования Фишера — Йейтса (вариант)**

```
for i = 1 to n do a[i] = i;
for i = 1 to (n - 1) do swap[a[i], a[Random[1, n]]];
```

Такие тонкости демонстрируют, почему нужно быть очень осторожным с алгоритмами генерации случайных чисел. Более того, я рекомендую подвергнуть достаточно длительному испытанию любой генератор случайных чисел, прежде чем по-настоящему доверять выдаваемым им результатам. Например, сгенерируйте четырехэлементные случайные перестановки 10 000 раз и убедитесь, что количество появлений всех 24 разных перестановок примерно одинаковое. Если же вы знаете, как проверить статистическую значимость, то сможете оценить надежность вашего алгоритма еще точнее.

### Реализации

Библиотека STL на языке C++ содержит две функции для генерирования перестановок в лексикографическом порядке: `next_permutation` и `prev_permutation`. На веб-сайте

**http://www.jjj.de/fxt** вы найдете коды на языке C++ для генерирования большого разнообразия комбинаторных объектов, включая перестановки и циклические перестановки.

Профессор Фрэнк Раски (Frank Ruskey) из Университета города Виктория в Канаде разработал *сервер комбинаторных объектов* (Combinatorial Object Server) для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://combos.org/**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Рекомендую посетить этой сайт и оценить предоставляемые им средства. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Книга [NW78] уже многие годы является ценным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации алгоритмов (на языке FORTRAN) для генерирования случайных перестановок и перестановок методом минимального изменения порядка. Кроме того, в ней приводятся процедуры для выявления циклов в перестановке. Подробности см. в *разд. 22.1.9*.

Библиотека Combinatorica (см. [PS03]) предоставляет реализации алгоритмов (на языке программирования пакета Mathematica) для генерирования случайных перестановок и последовательностей перестановок с минимальными различиями и в лексикографическом порядке. Эта библиотека также содержит процедуру поиска с возвратом для создания всех несовпадающих перестановок мультимножества и поддерживает разнообразные групповые операции над перестановками. Дополнительную информацию по Combinatorica см. в *разд. 22.1.8*.

### Примечания

Самым лучшим на сегодня справочным материалом по генерированию перестановок является одна из последних работ Дональда Кнута [Knu11]. Обзор, представленный в [Sed77], написан раньше, но прогресс в этой области весьма незначителен. Хорошие описания можно также найти в [KS99], [NW78] и [Rus03].

Методы быстрого генерирования перестановок выполняют только один обмен элементов местами для получения очередной перестановки. Алгоритм Джонсона — Троттера (см. [Joh63] и [Tro62]) удовлетворяет еще более строгому условию, а именно, что местами всегда обмениваются только смежные элементы. Простые функции с линейным временем исполнения для ранжирования перестановок описаны в работе [MR01].

В методах генерирования цепей Маркова произвольные объекты создаются с помощью произвольных преобразований — например, обмена элементов местами. Чтобы создать произвольную перестановку, достаточно выполнить $\Theta(n \log n)$ обменов местами элементов тождественной перестановки {1, 2, ..., n} — как рассматривается в анализе задачи о собирании купонов в *разд. 6.2.1*. Теория генерирования цепей Маркова излагается в работе [Sin12].

В прежние времена, когда компьютеры не были так распространены, многие пользовались таблицами случайных перестановок, печатаемых в специальных книгах, — например, в [MO63]. Рассмотренный ранее алгоритм для генерирования случайных перестановок методом обмена местами двух элементов перестановки впервые был описан в этой же книге.

### Родственные задачи

Генерирование случайных чисел (см. *разд. 16.7*), генерирование подмножеств (см. *разд. 17.5*), генерирование разбиений (см. *разд. 17.6*).

## 17.5. Генерирование подмножеств

**Вход.** Целое число $n$.

**Задача.** Сгенерировать: все возможные подмножества, или случайное подмножество, или очередное подмножество целых чисел $\{1, ..., n\}$ (рис. 17.5).

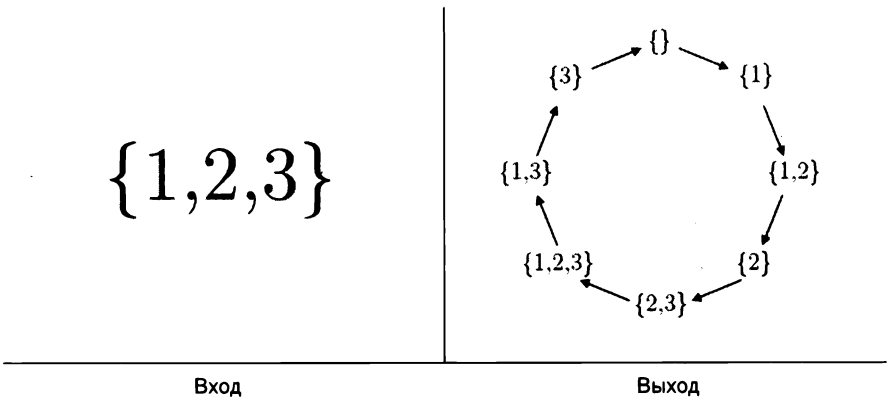



**Рис. 17.5.** Подмножества

### Обсуждение

*Подмножеством* называется выборка объектов, порядок расположения которых не имеет значения. Цель многих алгоритмических задач — найти наилучшее подмножество набора объектов. Например, в *задаче о вершинном покрытии* требуется найти наименьшее подмножество вершин, покрывающее все ребра графа, в *задаче о рюкзаке* — наиболее дорогостоящее подмножество объектов в пределах заданного общего веса, а в *задаче упаковки множества* — наименьшее подмножество подмножеств, которые в совокупности содержат ровно по одному вхождению каждого элемента.

Множество из $n$ элементов имеет $2^n$ разных подмножеств, включая пустое множество и само исходное множество. Количество подмножеств возрастает экспоненциально по отношению к значению $n$, но значительно медленнее, чем $n!$ перестановок из $n$ элементов. И действительно, поскольку $2^{20} = 1\,048\,576$, то задача полного перебора всех подмножеств множества из 20 элементов легко выполнима. Да и учитывая, что $2^{30} = 1\,073\,741\,824$, предел будет достигнут лишь при немного больших значениях $n$.

По определению подмножества относительный порядок его членов не играет роли при выяснении различий между подмножествами. Таким образом, подмножества {1, 2, 5} и {2, 1, 5} являются одинаковыми. Но всегда полезно поддерживать отсортированный, или *канонический*, порядок элементов подмножества, чтобы ускорить выполнение таких операций, как проверка двух подмножеств на идентичность.

Так же как и в случае с перестановками (см. *разд. 17.1*), принципиальным моментом в задаче генерирования подмножеств является установление числовой последовательности среди всех $2^n$ подмножеств. Для этого существуют три основных подхода:

- *лексикографический порядок.*

  Лексикографический порядок означает, что элементы отсортированы, и часто является наиболее естественным способом генерирования комбинаторных объектов. Восемь подмножеств множества {1, 2, 3} располагаются в лексикографическом порядке следующим образом: {}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3} и {3}. Но генерирование подмножеств в лексикографическом порядке оказывается на удивление трудной задачей. Поэтому не стоит использовать этот подход, если только у вас нет какой-либо важной причины для этого;

- *код Грея.*

  Особенно интересной последовательностью подмножеств является порядок с минимальными различиями, в котором соседние подмножества отличаются присутствием (или отсутствием) только одного элемента. Такое упорядочение подмножеств, называемое *кодом Грея* (Gray code), показано на рис. 17.5.

  Генерирование подмножеств в порядке кода Грея выполняется очень быстро, поскольку для этого существует удачная рекурсивная процедура: создаем код Грея $G_{n-1}$ из $n-1$ элементов, создаем второй код Грея в виде обращенной копии первого кода и добавляем $n$ к каждому подмножеству этой копии, затем выполняем конкатенацию всех подмножеств и получаем $G_n$. Чтобы лучше понять этот процесс, рассмотрите пример, показанный на рис. 17.5.

  Так как разница между подмножествами состоит только в одном элементе, то алгоритмы поиска методов исчерпывающего перебора на основе кодов Грея могут быть весьма эффективными. Например, при решении задачи вершинного покрытия таким методом нужно будет обновлять изменение в покрытии, удалив или добавив только одно подмножество;

- *двоичный счет.*

  Самый простой подход к генерированию подмножеств основан на том обстоятельстве, что каждое подмножество $S'$ определяется элементами множества $S$, входящими в $S'$. Мы можем представить подмножество $S'$ битовым вектором из $n$ элементов, в котором бит $i$ установлен в том случае, если $i$-й элемент множества $S$ входит в подмножество $S'$. Таким образом определяется взаимно однозначное соответствие между $2^n$ двоичными последовательностями длиной $n$ и $2^n$ подмножествами $n$ элементов. Для $n = 3$ метод двоичного счета генерирует подмножества в следующем порядке: {}, {3}, {2}, {2, 3}, {1}, {1, 3}, {1, 2}, {1, 2, 3}.

  Это двоичное представление является ключом к решению всех задач генерирования подмножеств. Чтобы генерировать все подмножества, мы просто ведем счет от 0 до $2^n - 1$. Для каждого целого числа последовательно маскируем биты и составляем подмножество из элементов, соответствующих установленным битам. Чтобы создать *следующее* или *предшествующее* подмножество, просто увеличиваем или соответственно уменьшаем на единицу число текущего подмножества. Процедура маскирования как раз и является функцией деранжирования (обратной ранжирова-

нию) подмножества, в то время как функция *ранжирования* создает двоичное число, в котором установленные биты соответствуют элементам $S$, после чего преобразует это двоичное число в целое число.

Чтобы создать случайное подмножество, можно попробовать сгенерировать случайное целое число в диапазоне от 0 до $2^n - 1$ и выполнить по нему операцию, обратную ранжированию. Но в зависимости от способа округления, применяемого в генераторе случайных чисел, некоторые подмножества могут никогда не появиться. Будет намного лучше бросить монету $n$ раз и по результату $i$-го броска решить, включать или нет элемент $i$ в подмножество. Бросок монеты можно надежно смоделировать, генерируя случайное действительное или очень большое целое число и проверяя, превышает ли оно центральное значение своего диапазона.

На практике необходимость генерирования подмножеств часто возникает в работе с двумя тесно связанными друг с другом структурами данных: $k$-подмножествами и строками.

- Вместо создания всех подмножеств нам могут требоваться только подмножества из $k$ элементов. Число таких подмножеств равно $\binom{n}{k}$, что значительно меньше, чем $2^n$, особенно для небольших значений $k$.

  Самый лучший способ конструирования всех $k$-подмножеств — генерирование их в лексикографическом порядке. Функция ранжирования основана на том обстоятельстве, что существует $\binom{n-f}{k-1}$ $k$-подмножеств, у которых наименьший элемент равен $f$. Отсюда мы можем определить наименьший элемент в $m$-м $k$-подмножестве $n$ элементов. Затем рекурсивно делаем то же самое с последующими элементами подмножества (подробности см. далее — в *подразделе «Реализации»*).

- Генерирование всех подмножеств равнозначно генерированию всех $2^n$ строк из значений *истина* и *ложь*. Те же самые основные методы применимы к задаче генерирования всех или случайных строк на алфавитах размером $\alpha$, за исключением случаев, когда общее количество строк будет равным $\alpha^n$.

### Реализации

На веб-сайте **http://www.jjj.de/fxt** приводятся коды на языке C++ для генерирования значительного количества разнообразных комбинаторных объектов, включая подмножества и k-подмножества (комбинации).

Профессор Фрэнк Раски (Frank Ruskey) из Университета города Виктория в Канаде разработал *сервер комбинаторных объектов* (Combinatorial Object Server) для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://combos.org/**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Рекомендую посетить этой сайт и оценить предоставляемые им средства. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Книга [NW78] уже многие годы является ценным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации алгорит-

мов (на языке FORTRAN) для создания случайных подмножеств и для генерирования подмножеств в порядке кода Грея и в лексикографическом порядке. Кроме того, в ней приводятся процедуры для создания случайных $k$-подмножеств и для генерирования подмножеств в лексикографическом порядке. Подробности см. в *разделе 22.1.9*.

Библиотека Combinatorica (см. [PS03]) предоставляет реализации алгоритмов (на языке программирования пакета Mathematica) для создания случайных подмножеств и для генерирования последовательностей подмножеств с помощью кода Грея, методом двоичного счета и в лексикографическом порядке. Эта библиотека также содержит процедуры для создания случайных $k$-подмножеств и для генерирования последовательностей подмножеств в лексикографическом порядке. Дополнительную информацию по Combinatorica см. в *разделе 22.1.8*.

#### *Примечания*

Самый лучший на сегодня справочный материал по генерированию подмножеств — это одна из последних работ Дональда Кнута [Knu11]. Хорошие описания можно найти в [KS99], [NW78] и [Rus03]. В книге [Wil89] приводится более свежая информация, чем в [NW78], в том числе в ней содержится подробное обсуждение современных проблем генерирования кода Грея.

Первоначально коды Грея были разработаны для обеспечения надежности передачи цифровых сигналов по аналоговому каналу (см. [Gra53]). При установке кодовых слов в порядке кода Грея $i$-е слово лишь слегка отличается от $(i + 1)$-го слова, так что незначительные колебания уровня аналогового сигнала искажают лишь небольшое количество битов. Коды Грея достаточно точно соответствуют гамильтоновым циклам на гиперкубе. В работе [Sav97] представлен отличный обзор кодов Грея (упорядочения по принципу минимального изменения) для большого класса комбинаторных объектов, включая подмножества. В работе [RW09] рассматривается интересный новый подход к генерированию $k$-подмножеств, основанный на сдвиге битов строк.

Популярная головоломка Spinout®, выпускаемая компанией ThinkFun (ранее носившей название Binary Arts Corporation), решается на основе идей кодов Грея.

### Родственные задачи

Генерирование перестановок (см. *разд. 17.4*), генерирование разбиений (см. *разд. 17.6*).

## 17.6. Генерирование разбиений

**Вход.** Целое число $n$.

**Задача.** Создать: все возможные разбиения целого числа или множества размером $n$, или случайное разбиение, или очередное разбиение (рис. 17.6).

### Обсуждение

Термином *разбиение* обозначаются два разных типа комбинаторных объектов: разбиения целых чисел и разбиения множеств. Вам необходимо понимать разницу между ними.

- *Разбиения целого числа n* представляют собой мультимножества ненулевых целых чисел, сумма которых равна $n$. Например, число 5 имеет семь разных разбиений:

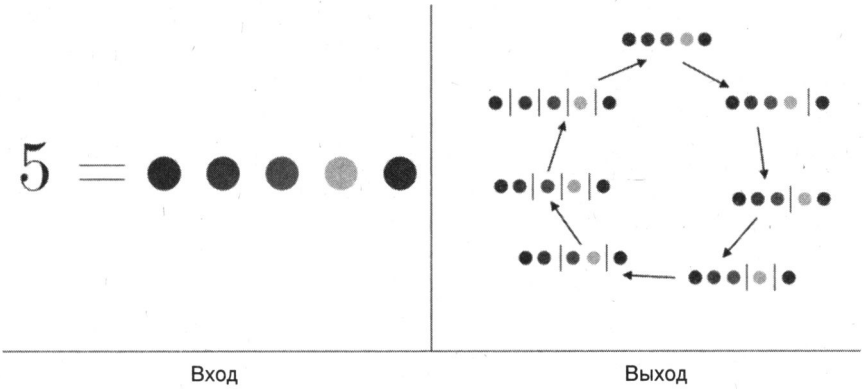



**Рис. 17.6.** Разбиения

{5}, {4,1}, {3,2}, {3,1,1}, {2,2,1}, {2,1,1,1} и {1,1,1,1,1}. Одним из интересных приложений, требующих генерирования разбиений целых чисел, с которым мне пришлось столкнуться, было моделирование расщепления ядра атома. При расщеплении ядро атома разбивается на множество кластеров меньшего размера. Общее количество частиц в этом множестве кластеров должно быть равным первоначальному количеству $n$ частиц ядра. Таким образом, целочисленные разбиения этого количества $n$ частиц представляют все возможные способы расщепления атома.

- *Разбиения множества* разделяют множество {1, ..., $n$} на непустые подмножества. Например, для множества $n = 4$ существует 15 разных разбиений: {1234}, {123,4}, {124,3}, {12,34}, {12,3,4}, {134,2}, {13,24}, {13,2,4}, {14,23}, {1,234}, {1,23,4}, {14,2,3}, {1,24,3}, {1,2,34} и {1,2,3,4}. Алгоритмические задачи, включая задачу *раскраски вершин/ребер* и задачу *поиска компонентов связности*, имеют в качестве результатов разбиения множества.

Для краткости мы станем далее называть разбиения целых чисел просто *разбиениями*, а для другого типа разбиений употреблять их полное название — *разбиение множеств*. А полное название: *разбиения целого числа* — будем применять в тех случаях, где нужно избежать недопонимания, какое именно из разбиений имеется в виду.

Количество разбиений возрастает экспоненциально по отношению к $n$, но, к счастью, с небольшим ускорением. Так, для целого числа $n = 20$ существует всего лишь 627 разбиений. Более того, возможно привести даже все разбиения числа $n = 100$, поскольку их количество равно всего лишь 190 569 292.

Самый простой способ создания разбиений — генерирование их в порядке, обратном лексикографическому. Первым разбиением будет само число {$n$}. Общее правило состоит в вычитании 1 из наименьшего элемента разбиения, большего единицы, с последующим объединением всех единиц с целью соответствия новому наименьшему элементу, большему единицы. Например, разбиением, следующим после {4,3,3,3,1,1,1,1}, будет {4,3,3,2,2,2,1}, поскольку пять единиц, получаемых после операции вычитания $3 - 1 = 2$, лучше всего объединяются в виде элемента {2,2,1}. Когда мы получаем разбиение, все члены которого являются единицами, то завершаем первый проход по всем разбиениям.

Этот алгоритм достаточно сложен для программирования, поэтому следует рассмотреть использование одной из готовых его реализаций, упоминаемых далее. В любом случае, проверьте, что используемая реализация выдает для $n = 20$ ровно 627 разных разбиений.

Задача генерирования случайных разбиений с равномерным распределением более сложна, чем аналогичная задача для перестановок или подмножеств. Это потому, что выбор первого (т. е. самого большого) элемента разбиения сильно влияет на количество разбиений, которые можно сгенерировать. Обратите внимание, что существует только одно разбиение $n$, наибольший элемент которого равен 1, а именно {1,1, ...,1}. Количество разбиений числа $n$ с наибольшим элементом, не превышающим $k$, задается следующим рекуррентным соотношением:

$$P_{n,k} = P_{n-k,k} + P_{n,k-1},$$

поскольку любое такое разбиение или содержит, или не содержит элемент с размером $k$. Для всех $y \le x$ имеются два граничных условия: $P_{n,1} = 1$ *и* $P_{x\ y,x} = P_{x\ y,x\ y}$. Второе условие может выглядеть странным, но оно правильное. Обратите внимание, что $P_{3,5}$ должно равняться $P_{3,3}$, т. к. не существует разбиений для 3, наибольший элемент которых равняется 4 или 5. С помощью приведенной функции можно выбрать наибольший элемент случайного разбиения с требуемой вероятностью, после чего рекурсивно создать случайное разбиение целиком.

При генерировании случайных разбиений наблюдается тенденция появления огромного количества сравнительно небольших элементов, как можно видеть на диаграмме Феррерса (Ferrers diagram), приведенной на рис. 17.7.

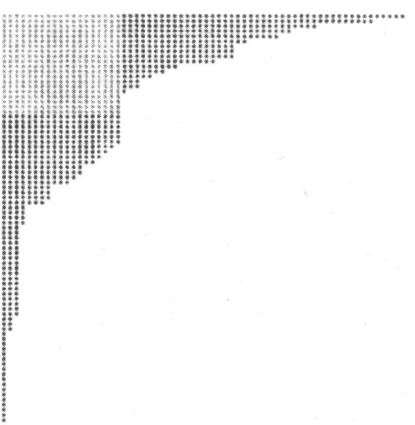


**Рис. 17.7.** Диаграмма Феррерса для случайного разбиения числа $n = 1000$

Каждая строка диаграммы соответствует одному элементу разбиения, упорядоченному по его размеру, который представлен в этой строке количеством точек. Такие диаграммы представляют собой хорошее средство для рассмотрения разбиений.

Одно из применений таких диаграмм состоит в оценке истории публикаций ученых-исследователей. Чем больше научная статья ученого цитируется в работах других ученых, тем более важной она считается. Значение H-индекса ученого равно $h$, если он

опубликовал как минимум $h$ научных статей, каждая из которых получила как минимум $h$ цитирований. Значение H-индекса соответствует размеру центрального квадрата, определяемого диаграммой Феррера для цитирований автора.

Разбиения множества можно генерировать, используя методы, подобные методам генерирования разбиений целых чисел. Каждое разбиение множества представляется в виде *функции ограниченного роста* $a_1, ..., a_n$, где $a_1 = 0$ и $a_i \leq 1 + \max(a_1, ..., a_{i-1})$ для $i = 2, ..., n$. Каждая цифра определяет подмножество (или *блок*) разбиения, в то время как условие роста обеспечивает сортировку блоков в каноническом порядке по наименьшему элементу в каждом блоке. Например, функция ограниченного роста 0, 1, 1, 2, 0, 3, 1 определяет такое разбиение множества: {{1, 5},{2, 3, 7},{4},{6}}.

Так как между разбиениями множеств и функциями ограниченного роста существует взаимно однозначное соответствие, мы можем воспользоваться лексикографическим порядком функций ограниченного роста для упорядочивания разбиений множеств. В самом деле, 15 разбиений множества {1,2,3,4}, упомянутх в определении разбиения множества в начале обсуждения, расположены в соответствии с лексикографическим порядком их функций ограниченного роста (вы можете в этом убедиться).

Для генерирования случайных разбиений множества можно использовать метод двоичного счета, аналогичный используемому для генерирования разбиений целых чисел. Числами Стирлинга второго рода $\left\{ {n \atop k} \right\}$ называется количество разбиений $n$-элементного множества $\{1, ..., n\}$ на $k$ непустых подмножеств. Для их вычисления используется следующее рекуррентное соотношение:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

с краевыми условиями $\left\{ {n \atop n} \right\} = \left\{ {n \atop 1} \right\} = 1$. Подробности см. в последующих подразделах.

### Реализации

На веб-сайте **http://www.jjj.de/fxt** в пакете комбинаторики приведены коды на языке C++ для генерирования значительного количества разнообразных комбинаторных объектов, включая разбиения и композиции (разложения) целых чисел. Реализации, описываемые в книге [KS99], генерируют разбиения как целых чисел, так и множеств в лексикографическом порядке, включая функции ранжирования. Реализации этих генераторов на языке C можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Профессор Фрэнк Раски (Frank Ruskey) из Университета города Виктория в Канаде разработал *сервер комбинаторных объектов* (Combinatorial Object Server) для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://combos.org/**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Рекомендую посетить этой сайт и оценить предоставляемые на нем средства. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Книга [NW78] уже многие годы является замечательным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации (на

языке FORTRAN) алгоритмов для генерирования случайных и последовательных разбиений целых чисел, множеств и таблиц Юнга. Подробности см. в *разд. 22.1.9*.

Библиотека Combinatorica (см. [PS03]) содержит реализации алгоритмов (на языке программирования пакета Mathematica) для генерирования случайных и последовательных разбиений целых чисел, строк и таблиц Юнга, а также средства для подсчета и манипулирования этими объектами. Подробности см. в *разд. 22.1.8*.

#### Примечания

Самым лучшим на сегодня справочным материалом по алгоритмам для генерирования разбиений как целых чисел, так и множеств, является одна из последних работ Дональда Кнута [Knu11]. Хорошие описания можно также найти, например, в [KS99], [NW78], [Rus03] и [PS03]. Основным справочником по разбиениям целых чисел и родственным темам является книга [And98], а книга [AE04] представляет собой доступное введение в эту область. Одна из недавних работ на тему разбиения множеств — книга [Man12].

Разбиения целых чисел и множеств являются частными случаями разбиений *мультимножеств* или разбиений множеств с не обязательно разными элементами. В частности, разные разбиения мультимножества {1,1,1, ...,1} в точности соответствуют разбиениям целых чисел. Разбиения мультимножеств обсуждаются в работе Дональда Кнута [Knu11].

Длинная история генерирования комбинаторных объектов подробно излагается в работе Дональда Кнута [Knu11]. Особенно интересны связи между разбиениями множеств и японским ритуалом сжигания благовоний, а также между всеми 52 разбиениями множества для $n = 5$ и отдельными главами старинного японского романа «Повесть о Гэндзи».

В 2015 году на экраны вышел фильм «Человек, который познал бесконечность» — о жизни выдающегося индийского математика Рамануджана, рассказывающий в том числе о создании им его удивительной формулы для приблизительного подсчета количества разбиений целых чисел.

Родственными комбинаторными объектами являются таблицы Юнга и композиции целых чисел, хотя их возникновение в реальных приложениях маловероятно. Алгоритмы генерирования объектов обоих типов приведены в [NW78], [Rus03] и [PS03].

*Таблицы Юнга* представляют собой двумерные формирования целых чисел в диапазоне $\{1, ..., n\}$, где количество элементов в каждой строке определяется одним из разбиений целого числа $n$. Элементы каждой строки и каждого столбца отсортированы в возрастающем порядке, а строки выровнены по левому краю. Это понятие охватывает широкий набор разнообразных структур в виде частных случаев. Они обладают многими интересными свойствами, включая взаимно однозначное соответствие между парами таблиц и перестановок.

Композиции целых чисел представляют всевозможные способы распределения множества из $n$ одинаковых шаров по $k$ различным урнам. Например, три шара можно разместить по двум урнам таким образом: {3,0}, {2,1}, {1,2} или {0,3}. Композиции легче всего генерировать последовательно в лексикографическом порядке. Для генерирования случайных композиций выбираем случайное $(k-1)$-подмножество из $n+k-1$ элементов с помощью алгоритма, описанного в *разд. 17.5*, после чего подсчитываем количество оставшихся элементов между выбранными. Например, для $k = 5$ и $n = 10$, $(5-1)$-подмножество {1,3,7,14} множества $\{1, ..., (n+k-1) = 14\}$ определяет композицию {0,1,3,6,0} — поскольку нет элементов ни слева от элемента 1, ни справа от элемента 14.

### Родственные задачи

Генерирование перестановок (см. *разд. 17.4*), генерирование подмножеств (см. *разд. 17.5*).

# 17.7. Генерирование графов

**Вход.** Параметры, описывающие граф, включая количество вершин $n$ и количество ребер $m$ или вероятность наличия ребра $p$.

**Задача.** Сгенерировать: все графы, удовлетворяющие заданным параметрам, или случайный граф, или очередной граф (рис. 17.8).

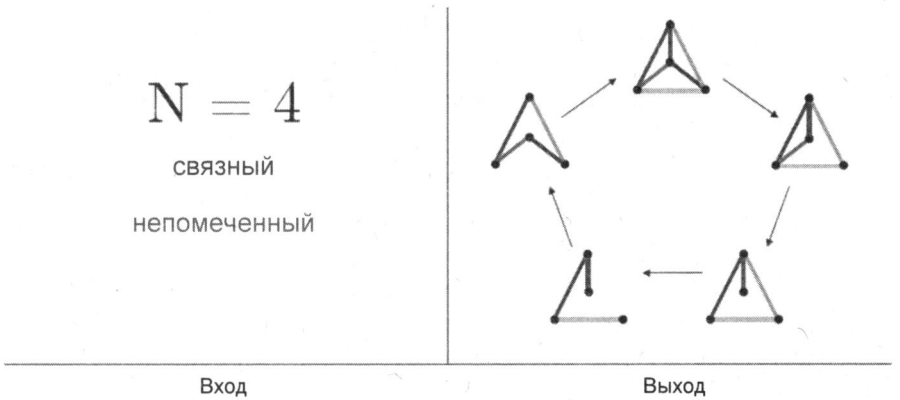



**Рис. 17.8.** Генерирование графов

### Обсуждение

Задача генерирования графов обычно возникает при создании тестовых данных для программ. Возможно, у вас имеются две разные программы для решения одной и той же задачи, и вы хотите узнать, какая из них работает быстрее, или удостовериться, что обе дают одинаковый (предположительно правильный) результат. Другим приложением задачи генерирования графов является проверка, обладают ли те или иные графы определенным свойством, или насколько оно распространено среди них. В справедливость теоремы четырех цветов легче поверить после демонстрации четырехцветной раскраски всех планарных графов на 15 вершинах.

Задача генерирования графов усложняется многими факторами. Прежде всего, нужно точно знать, граф какого типа требуется создать. Некоторые важные разновидности графов показаны на рис. 7.2 и описаны в сопровождающем его тексте. При генерировании графов необходимо ответить на следующие вопросы.

- *Требуются помеченные или непомеченные графы?*

  Иными словами, имеют ли значение при сравнении графов названия вершин. При генерировании помеченных графов мы создаем все возможные маркировки для всех возможных топологий графов. А при генерировании непомеченных графов мы хотим создать ровно по одному представителю каждой топологии и можем игнорировать метки. Например, из трех вершин можно создать только два связных непомеченных графа: треугольник и простой путь. Но из тех же трех вершин можно создать четыре связных помеченных графа: один треугольник и три трехвершинных пути, отличающиеся друг от друга названием центральной вершины. В общем, на-

много легче создавать помеченные графы. Но, скорее всего, мы при этом будем очень быстро завалены многими изоморфными копиями одних и тех же нескольких графов.

- *Требуются ориентированные или неориентированные графы?*

  Большинство естественных алгоритмов генерируют неориентированные графы. Эти графы можно преобразовать в ориентированные, выполнив ориентацию ребер по броску монеты. Из любого графа можно сделать бесконтурный ориентированный граф, упорядочив случайным образом вершины в линию и направив каждое ребро слева направо. При таком разнообразии вариантов вы должны все обдумать и решить, генерируются ли все графы единообразно и случайным образом, а также насколько важен для вас способ их генерирования.

Кроме прочего, нужно определиться с тем, что вы считаете случайным графом. Существуют три основные модели случайных графов, и все они генерируют графы согласно разным вероятностным распределениям:

- *генерирование случайных ребер.*

  Параметры модели Эрдёша — Реньи задаются вероятностью наличия ребра $p$. Для каждой пары вершин $x$ и $y$ решение о добавлении ребра $(x, y)$ принимается по результату броска монеты. При $p = 1/2$ будут сгенерированы с одинаковой вероятностью все *помеченные* графы, хотя для генерирования случайных разреженных графов можно использовать меньшие значения $p$;

- *выбор случайных ребер.*

  Параметры этой модели задаются требуемым количеством ребер $m$. Модель генерирует граф, выбирая $m$ разных ребер случайным образом с равномерным распределением. Один из способов сделать это — создание случайных пар вершин $(x, y)$ и соединение их ребрами, если какая-либо вершина еще не в графе. Есть и альтернативный подход: создать набор из $\binom{n}{2}$ возможных ребер и случайным образом выбрать из них $m$-подмножество, как показано в *разд. 17.5*;

- *избирательное присоединение* (preferential attachment).

  Согласно модели «богатые становятся богаче» более вероятно, что новые ребра будут направлены к вершинам высокого уровня, а не низкого. Посмотрите, как добавляются новые ссылки (ребра) к графу веб-страниц. По любой реалистичной сетевой модели генерирования очередная ссылка с большой степенью вероятности будет вести на Google, чем на **https://www.algorist.com**[1]. Выбор следующей соседней вершины с вероятностью, пропорциональной ее степени, порождает графы со свойствами степенной зависимости, которыми обладают многие реальные сети.

Какая из этих моделей подходит лучше всего для вашего приложения? Скорее всего, никакая. Случайные графы плохо структурированы по определению. И хотя графы часто используются для моделирования высокоструктурированных взаимоотношений,

[1] Пожалуйста, создайте ссылку с вашей домашней страницы на наш веб-сайт, чтобы исправить эту несправедливость.

интересные и легко выполнимые эксперименты на случайных графах, как правило, не отражают реальное положение вещей.

Альтернативой случайным графам являются «органические» графы, которые отображают взаимоотношения между реальными объектами. В Интернете имеется много источников взаимоотношений, которые можно превратить в интересные «органические» графы, приложив немного усилий по их программированию. Возьмем, например, граф, определяемый набором веб-страниц, в котором ребро задается гиперссылкой с одной страницы на другую. Или другой пример — граф, определяемый железными дорогами или авиарейсами, где вершины представляют станции или аэропорты, а ребра — прямую связь между ними.

Особенно интересные алгоритмы генерирования имеют два класса графов:

- *деревья.*

  Коды Прюфера (Prufer codes) предоставляют простой способ ранжирования *помеченных* деревьев и, таким образом, решения всех стандартных задач генерирования (см. *разд. 17.4*). Для $n$ вершин существует ровно $n^{n-2}$ помеченных деревьев и столько же строк длиной $n-2$ существует на алфавите $\{1, 2, ..., n\}$.

  Ключом к взаимно однозначному соответствию Прюфера является то обстоятельство, что каждое дерево имеет по крайней мере две вершины степени 1. То есть в любом помеченном дереве вершина $v$, соединенная с листом с наименьшей меткой, является четко определенной. Пусть $S_1$ — первый символ кода — это вершина $v$. Удалим связанный лист и будем повторять эту процедуру до тех пор, пока не останутся только две вершины. Таким путем мы получим уникальный код $S$ для любого помеченного дерева, который можно использовать для ранжирования дерева. Чтобы построить дерево по коду, воспользуемся тем обстоятельством, что листом с наименьшей меткой будет целое число, отсутствующее в $S$, которое совместно с $S_1$ определяет первое ребро дерева. Применяя индукцию, получаем все дерево;

- *графы с фиксированной степенной последовательностью.*

  *Степенной последовательностью* (degree sequence) графа $G$ называется разбиение целого числа $p = (p_1, ..., p_n)$, где $p_i$ является степенью $i$-й вершины с наивысшей степенью графа. Так как каждое ребро учитывается при подсчете степени двух вершин, то $p$ является разбиением целого числа $2m$, где $m$ представляет собой количество ребер в графе.

  Не все разбиения соответствуют степенным последовательностям графов. Но существует рекурсивный метод, который создает граф для заданной степенной последовательности, если такой граф вообще существует. Если разбиение выполнимо, то вершину с наивысшей степенью $v_1$ можно соединить с вершинами со степенями, наивысшими из оставшихся, или вершинами, соответствующими частям $p_2,...,p_{p_1+1}$.

  Удалив $p_i$ и уменьшив $p_2,...,p_{p_1+1}$ на единицу, мы получаем меньшее разбиение для рекурсивной обработки. Если мы завершим процедуру, не создав отрицательных чисел, то разбиение будет осуществлено. Так как мы всегда соединяем вершину наивысшей степени с другими вершинами высокой степени, то важно после каждой итерации переупорядочить элементы разбиения по размеру.

Хотя этот метод является детерминистским, с его помощью из графа $G$ можно создать полуслучайную коллекцию графов, используя операции обмена ребер. Допустим, что ребра $(x, y)$ и $(w, z)$ входят в граф $G$, а ребра $(x, w)$ и $(y, z)$ — нет. Обменяв эти пары ребер, мы получим другой (не обязательно связный) граф, не меняя степень ни одной из вершин.

### Реализации

База графов Stanford GraphBase (см. [Knu94]) является, пожалуй, самым хорошим генератором экземпляров графов для использования их в качестве тестовых данных для других программ. Поскольку в ней задействуются аппаратно-независимые генераторы случайных чисел, созданные с ее помощью случайные графы можно воспроизвести на других машинах, что позволяет использовать их в качестве входных данных при экспериментальном сравнении алгоритмов. Дополнительную информацию см. в *разд. 22.1.7*.

Ресурсы для реальных сетей предоставляют, среди прочих, Network Data Repository (**http://networkrepository.com/**) и Stanford Large Network Dataset Collection (**https://snap.stanford.edu/data/**). Посетите эти веб-сайты и посмотрите, что они могут предложить вам.

Библиотека Combinatorica (см. [PS03]) содержит генераторы (на языке пакета Mathematica) для графов типа «звезда» и «колесо», для полного графа, для случайных графов и деревьев, а также для графов с заданной степенной последовательностью. Кроме этого, библиотека включает в себя операции создания более сложных графов на основе упомянутых, в том числе объединение и произведение графов.

Сервер комбинаторных объектов (**http://combos.org/**) предоставляет процедуры для генерирования как свободных, так и корневых деревьев.

Программа Nauty для тестирования графов на изоморфизм (см. *разд. 19.9*) содержит набор программ для генерирования неизоморфных графов, а также специальные генераторы для создания двудольных графов, ориентированных графов и мультиграфов. Все эти программы можно загрузить с веб-страницы **http://users.cecs.anu.edu.au/~bdm/nauty/**.

Математик Брендан Маккей (Brendan McKay) приводит на своей веб-странице (**http://cs.anu.edu.au/~bdm/data**) полные каталоги нескольких семейств графов и деревьев. На веб-сайте House of Graphs (**https://hog.grinvin.org/**) содержится набор тщательно подобранных графов с интересными свойствами, предназначенных для генерирования контрпримеров для гипотез. Дополнительную информацию см. в [BCGM13].

В книге [NW78] описаны эффективные процедуры (на языке FORTRAN) перечисления всех помеченных деревьев с помощью кодов Прюфера и создания случайных непомеченных корневых деревьев. Подробности см. в *разд. 22.1.9*. В книге [KS99] описывается алгоритм на языке C для генерирования помеченных деревьев. Реализации можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

#### Примечания

На тему генерирования случайных графов с равномерным распределением написано множество книг. Среди прочих обзоров можно назвать [Gol93] и [Tin90]. В работе [NLKB11] демонстрируется использование графических процессоров для быстрого генерирования

случайных графов. С задачей генерирования классов графов тесно связана задача их подсчета. Обзор достижений в этой области приведен в книге [HP73].

Самым лучшим на сегодня справочным материалом по генерированию деревьев является одна из последних работ Дональда Кнута [Knu11]. Взаимно однозначное соответствие между $(n-2)$-строками и помеченными деревьями было установлено немецким математиком Хайнцем Прюфером (Heinz Prufer) в 1918 году (см. [Pru18]).

В теории случайных графов пороговые законы определяют плотность ребер, при которой становится высокой вероятность существования таких свойств, как связность. Изложение теории случайных графов можно найти в книгах [Bol01], [FK15] и [JLR00].

Модель избирательного присоединения графов возникла сравнительно недавно при изучении сетей. Введение в эту интересную область содержится в книгах [Bar03] и [Wat04]. В книгах [BD11] и [VL05] описаны методы генерирования графов с заданными последовательностями степеней.

Разбиение целого числа является *графическим*, если существует простой граф с такой степенной последовательностью. В работе [EG60] приводится доказательство, что степенная последовательность является графической тогда и только тогда, когда для любого целого числа $r < n$ соблюдается следующее условие:

$$\sum_{i=1}^{r} d_i \le r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i).$$

### Родственные задачи

Генерирование перестановок (см. *разд. 17.4*), изоморфизм графов (см. *разд. 19.9*).

## 17.8. Календарные вычисления

**Вход.** Календарная дата $d$, заданная месяцем, днем и годом.

**Задача.** На какой день недели выпадает $d$ в этой календарной системе (рис. 17.9)?

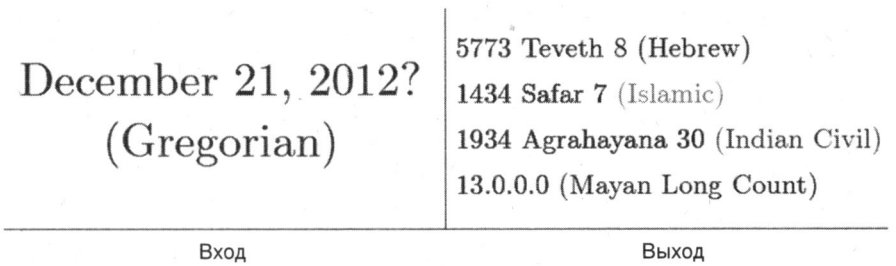



**Рис. 17.9.** Календарные вычисления

### Обсуждение

В бизнес-приложениях часто требуется выполнять календарные вычисления. Например, вывести календарь для определенного месяца и года или вычислить, в какой день недели или года произойдет какое-то событие. Важность календарных вычислений была ярко продемонстрирована «ошибкой 2000 года», присутствовавшей в старых программах, где для обозначения года использовались только две последние цифры.

Более сложные вопросы возникают в международных приложениях, поскольку разные народы и этнические группы используют разные календарные системы. Некоторые из этих календарных систем — такие как используемый в большинстве стран мира григорианский календарь — основаны на солнечном цикле, в то время как другие календари — например, иудейский календарь, основаны на лунных циклах. А смогли бы вы назвать сегодняшнюю дату по китайскому календарю?

Календарные вычисления отличаются от других задач, рассматриваемых в этой книге, потому что календари являются историческими объектами, а не математическими. В области календарных вычислений алгоритмические вопросы касаются установления правил календарной системы и правильной их реализации, а не разработки эффективных методов вычисления.

Подход, лежащий в основе всех календарных систем, заключается в выборе начальной точки и ведении отсчета от этой точки. Разные календарные системы отличаются друг от друга специфическими правилами, определяющими конец одного месяца или года и начало другого. Для реализации календаря требуются две функции: одна из них по заданной дате возвращает количество дней, прошедших от начальной точки, а другая — по заданному целому числу $n$ возвращает дату календаря, отстоящую ровно на $n$ дней от точки начала отсчета. Эти функции аналогичны функциям ранжирования для комбинаторных объектов — таких как перестановки (см. *разд. 17.4*).

Главным источником проблем в календарных системах является то обстоятельство, что в солнечном году не целое число дней. Чтобы поддерживать синхронизацию календарного года с реальным, время от времени в него нужно вносить поправки в виде високосных дней. Но т. к. продолжительность солнечного года составляет 365 дней, 5 часов, 49 минут и 12 секунд, то добавление високосного дня через каждые четыре года делает корректируемый календарный год на 10,8 минуты длиннее.

В первоначальном юлианском (от Юлия Цезаря) календаре эти лишние минуты не принимались во внимание, и к 1582 году накопилось отставание в 10 дней. Тогда Папа Григорий XIII ввел в действие исправленный календарь, названный по его имени григорианским, который используется по сей день. Исправления состояли в одномоментном переносе даты календаря на 10 дней вперед, а также правилах добавления високосных дней в года, кратные четырем, и устранения високосных дней в годах, кратных 100, но не 400. По некоторым сведениям, введение нового календаря сопровождалось бунтами среди населения — люди считали, что их жизнь укоротилась на десять дней. Вне пределов влияния католической церкви принятие нового календаря проходило медленно. В Англии и Америке григорианский календарь был принят в 1752 году, а в Турции — аж в 1927-м.

Правила для большинства календарных систем достаточно сложны, вследствие чего вполне разумно воспользоваться уже готовым кодом из надежного источника вместо того, чтобы пытаться писать свою реализацию.

Существует много алгоритмов типа «удиви своих друзей», используя которые можно в уме высчитать день недели для определенной даты. Но такие алгоритмы часто работают правильно только в определенном столетии, и их не следует реализовывать на компьютере.

### Реализации

Библиотеки процедур для работы с календарями широко доступны как на языке C, так и Java. Библиотека Boost предоставляет надежную реализацию григорианского календаря на языке C++. Код ее можно загрузить с веб-страницы **https://www.boost.org/doc/libs/1_70_0/ doc/html/date_time.html**. Стандартный пакет утилит Java java.util содержит класс `GregorianCalendar`, производный от абстрактного суперкласса `Calendar`, который реализует григорианский календарь. Любой из этих реализаций будет, скорее всего, достаточно для большинства приложений.

В книге [RD18] описаны алгоритмы для разных календарей, включая григорианский, китайский, индийский, мусульманский и иудейский, а также календари, представляющие исторический интерес. Реализация этих календарей на языке Common Lisp, Java и Mathematica называется Calendrical и содержит процедуры преобразования дат между разными календарными системами, вычисления дня недели, а также определения светских и религиозных праздников. Пакет Calendrical является, пожалуй, самым обширным и надежным источником существующих календарных процедур. Подробности см. на веб-сайте **http://calendarists.com**.

Реализации интернациональных календарей, написанные на языках C и Java, доступны на веб-сайте GitHub (**https://github.com/**), но об их надежности ничего не известно. Чтобы найти требуемый календарь, выполните поиск по соответствующему ключевому слову — например, `Gregorian calendar` — для григорианского календаря.

#### *Примечания*

Всесторонний обзор алгоритмов для календарных вычислений представлен в работах [DR90] и [RDC93], на основе которых была написана книга [DR18], содержащая алгоритмы по крайней мере для 25 национальных и исторических календарей. В книге [DR02] представлены календарные таблицы на 300 лет — с 1900 года по 2200-й.

### Родственные задачи

Арифметические операции с произвольной точностью (см. *разд. 16.9*), генерирование перестановок (см. *разд. 17.4*).

## 17.9. Календарное планирование

**Вход.** Бесконтурный ориентированный граф $G = (V, E)$, в котором вершины представляют задания, а ребро $(u, v)$ отражает тот факт, что задание $u$ должно быть завершено перед заданием $v$.

**Задача.** Составить такое расписание выполнения всех заданий, которое минимизирует время выполнения или количество процессоров (рис. 17.10).

### Обсуждение

Задача разработки наилучшего расписания, удовлетворяющего набору ограничивающих условий, является фундаментальной для многих приложений. С ней тесно связана задача распределения заданий между процессорами, представляющая собой критический аспект любой системы параллельной обработки данных. Плохое планирование может привести к простаиванию многих процессоров — кроме того, который выпол-

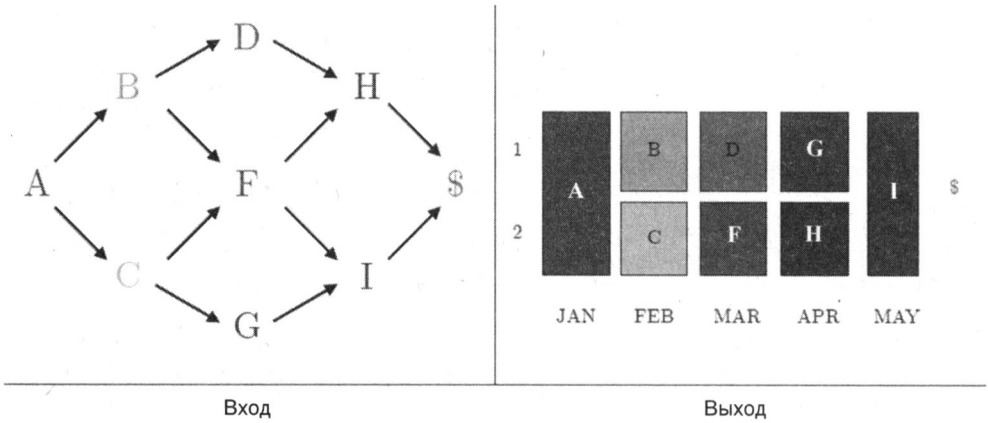



**Рис. 17.10.** Задача календарного планирования

няет задание, представляющее собой «узкое место». В качестве других примеров задач календарного планирования можно привести выдачу заданий рабочим, размещение студенческих групп по аудиториям или составление расписания экзаменов.

Задачи календарного планирования различаются по типам ограничивающих условий и видам составляемых расписаний. При этом к вариантам календарного планирования имеют отношение и некоторые из рассматриваемых в этой части книги задач. Говоря более конкретно:

- *топологическая сортировка* — создает расписание, отвечающее ограничивающим условиям очередности в бесконтурном ориентированном графе. Подробности см. в *разд. 18.2*;
- *паросочетание в двудольных графах* — сопоставляет задания с рабочими, имеющими навыки, требуемые для их выполнения. Подробности см. в *разд. 18.6*;
- *раскраска ребер и вершин* — сопоставляет задания с временными периодами таким образом, чтобы избежать попадания конфликтных заданий в один и тот же временной интервал. Подробности см. в *разд. 19.7* и *19.8*;
- *алгоритм коммивояжера* — выбирает оптимальный маршрут для посещения указанных адресов разносчиком пиццы. Подробности см. в *разд. 19.4*;
- *алгоритм поиска эйлерова цикла* — создает наиболее эффективный путь следования снегоуборочной машины, проходящей по всем заданным улицам. Подробности см. в *разд. 18.7*.

Рассмотрим задачу календарного планирования с условиями очередности для бесконтурных ориентированных графов. Допустим, что вы разбили большую работу на несколько меньших заданий. Для каждого задания известно время, требуемое для его выполнения. Кроме этого, для каждой пары заданий *A* и *B* известно, существует ли необходимость, чтобы задание *A* выполнялось перед заданием *B*. Чем меньше ограничивающих условий, тем лучшее расписание можно получить. Эти ограничивающие условия должны определять бесконтурный ориентированный граф, поскольку цикл при ограничивающих условиях очередности создает конфликт, который невозможно разрешить.

Интерес представляют следующие задачи:

- *критический путь.*

  Это самый длинный путь от начальной вершины до конечной. Знать его величину очень важно, поскольку единственным осуществимым способом сократить минимальное возможное время выполнения проекта будет сокращение длительности задания в каждом критическом пути. Критический (самый длинный) путь в бесконтурном ориентированном графе можно определить за время $O(n + m)$, используя методы динамического программирования;

- *минимальное время завершения.*

  Самое короткое время, за которое можно завершить проект, соблюдая при этом условия очередности и предполагая наличие неограниченного количества исполнителей. При отсутствии условий очередности все задания можно было бы выполнять одновременно, а общее время было бы равно времени, требуемому для выполнения самого длительного из них. А если на все отдельные задания наложены строгие ограничивающие условия, согласно которым каждое задание должно выполняться только после выполнения его непосредственного предшественника, то минимальное время завершения будет равно сумме времен выполнения всех заданий.

  Для бесконтурного ориентированного графа минимальное время выполнения можно вычислить за время $O(n + m)$, т. к. оно зависит от критического пути. Чтобы составить расписание, перебирайте задания в топологическом порядке. Каждое задание должно запускаться на новом процессоре, как только завершится выполнение непосредственно предшествующего ему задания;

- *каков компромисс между количеством исполнителей и временем завершения?*

  Мы заинтересованы в быстрейшем выполнении работы при заданном количестве исполнителей. К сожалению, эта и большинство подобных ей задач являются NP-полными.

Реальные приложения календарного планирования часто имеют ограничивающие условия, смоделировать которые посредством описанных методов будет трудно или невозможно. Как, например, не допустить совместную работу двух сотрудников, недолюбливающих друг друга? Существуют два приемлемых подхода к решению подобных задач. В первом случае мы можем игнорировать такие неявные ограничивающие условия до самого конца, а затем подправить расписание, чтобы оно удовлетворяло этим ограничениям. Другой подход — задачу календарного планирования можно сформулировать во всей ее сложности в терминах линейного целочисленного программирования (см. *разд. 16.6*). Я рекомендую сначала применить самый простой способ и посмотреть, что из этого получится, и только потом переходить к более сложным методам.

Другая фундаментальная задача календарного планирования заключается в распределении набора заданий без ограничивающих условий очередности между идентичными исполнителями таким образом, чтобы минимизировать общее время исполнения. Например, в типографии требуется распределить несколько работ между $k$ копировальными машинами так, чтобы завершить работу к концу текущего рабочего дня. Эту за-

дачу можно смоделировать как задачу разложения по контейнерам (см. *разд. 20.9*), где каждому заданию присваивается числовое значение, равное количеству часов, требуемому для ее завершения, а каждая машина представляется контейнером с емкостью, равной количеству рабочих часов.

В более сложных вариантах задачи календарного планирования для каждого задания указывается время, раньше которого нельзя начинать выполнение, и крайний срок завершения. Для решения задач календарного планирования существуют эффективные эвристические алгоритмы, основанные на сортировке заданий по размеру и требуемому времени завершения. Дополнительную информацию о них вы найдете в *подразделах «Реализации»* и *«Примечания»*. Обратите внимание на то, что эти задачи календарного планирования становятся сложными только в том случае, если задания нельзя распределить по нескольким машинам или прерывать их исполнение с последующим возобновлением. Если ваше приложение дает вам высокую степень свободы, то ею следует воспользоваться.

### Реализации

Коллекция программ JOBSHOP на языке C представляет собой средство для решения задач календарного планирования, разработанное на основе исследований, содержащихся в работе [AC91]. Коллекцию можно загрузить с веб-сайта **http://www.math.uwaterloo.ca/~bico//jobshop/**.

Комплексная система календарного планирования UniTime для учебных заведений (**https://www.unitime.org/**) обеспечивает разработку расписаний занятий и экзаменов, а также распределение студентов по отдельным курсам. Система предоставляется по лицензии на открытое программное обеспечение.

Программа календарного планирования LEKIN (см. [Pin16]) предназначена для образовательный целей. Она поддерживает планирование как для одной, так и для нескольких машин. Загрузить программу можно с веб-сайта **http://www.stern.nyu.edu/om/software/lekin/**.

Программа ILOG CP предоставляет для коммерческих приложений планирования наиболее современный уровень технологий в рассматриваемой области (см. [LRSV18]). Дополнительную информацию можно найти на веб-сайте **https://www.ibm.com/analytics/cplex-cp-optimizer**. Также доступна бесплатная версия этой программы с ограниченными возможностями.

#### *Примечания*

Существует огромное количество литературы об алгоритмах планирования. Подробный обзор достижений в этой области дается в книгах [Bru07] и [Pin16]. Подборка свежих обзоров по всем аспектам календарного планирования приводится в книге [LA04]. А в книге [But11] рассматривается календарное планирование реального времени для вычислительных систем.

Разработана хорошо определенная классификация нескольких тысяч вариантов расписаний по машинной среде, по особенностям обработки и ограничивающим условиям, а также по параметрам, подлежащим минимизации. Обзоры календарного планирования реального времени для вычислительных систем содержатся в книгах [Bru07], [CPW98], [LLK83] и [Pin16].

*Диаграммы Ганта* дают визуальные представления решений распределения работ по машинам, где по оси абсцисс отложено время, а разные машины отображаются в разных рядах. Пример диаграммы Ганта приведен на рис. 17.10, где каждая запланированная работа показана в виде прямоугольника, определяющего ее исполнителя и время начала и окончания. Методы календарного планирования проектов с ограничивающими условиями очередности часто называют системой PERT/CPM (Program Evaluation and Review Technique/Critical Path Method — система оценки и пересмотра планов / метод критического пути). Как диаграммы Ганта, так и система PERT/CPM обсуждаются в большинстве учебников по исследованию операций, включая [Pin16].

При рассмотрении задач календарного планирования уроков и других подобных задач часто используется термин *составление расписания* (timetabling). На проводящейся дважды в год конференции PATAT (Practice and Theory of Automated Timetabling, практика и теория автоматического составления расписания) обсуждаются новые результаты в этой области. Дополнительная информация доступна на веб-сайте **https://patatconference.org/**.

### Родственные задачи

Топологическая сортировка (см. *разд. 18.2*), паросочетание (см. *разд. 18.6*), вершинная раскраска (см. *разд. 19.7*), реберная раскраска (см. *разд. 19.8*), разложение по контейнерам (см. *разд. 20.9*).

## 17.10. Выполнимость

**Вход.** Набор дизъюнкций в конъюнктивной нормальной форме.

**Задача.** Определить, существует ли такой набор значений истинности булевых переменных, для которого одновременно выполняются все дизъюнкции (рис. ЦВ-17.11).

### Обсуждение

Задача выполнимости (Satisfiability, SAT) возникает в ситуации, когда нужно найти конфигурацию или объект, который должен удовлетворять набору логических ограничений. Задача выполнимости часто используется для проверки того, что проектируемая аппаратная или программная система работает должным образом для всех входных экземпляров. Допустим, что логическая формула $S(X)$ обозначает указанный результат на входных переменных $X = x_1, ..., x_n$, а формула $C(X)$ обозначает булеву логику предлагаемой схемы вычисления $S(X)$. Схема будет правильной в том случае, если нет такого значения $\bar{X}$, для которого $S(\bar{X}) \neq C(\bar{X})$.

Задача выполнимости представляет собой исходную NP-полную задачу. Хотя она задействуется на практике в таких областях, как выполнимость ограничивающих условий, логика и автоматическое доказательство теорем, она имеет большую теоретическую значимость как базовая задача, на основе которой строятся все другие доказательства NP-полноты. В создание лучших из современных систем решения задач SAT было вложено столько усилий, что их следует использовать в любом случае, когда вам действительно нужно *точно* решить NP-полную задачу. Однако лучшим подходом к решению NP-полных задач обычно является применение эвристических алгоритмов, которые дают хорошие, хотя и не оптимальные результаты.

При проверке выполнимости необходимо найти ответы на следующие вопросы.

- *В какой форме представлена ваша формула: «И из всех ИЛИ» или «ИЛИ их всех И»?*

  В задаче выполнимости ограничивающие условия указываются в виде логической формулы. Существуют два основных способа выражения логических формул: в конъюнктивной нормальной форме (КНФ) и дизъюнктивной нормальной форме (ДНФ). Формулы в КНФ состоят из конъюнкций (операция И) и дизъюнкций (операция ИЛИ), как в следующем примере:

  $$(v_i \text{ ИЛИ } \bar{v}_2) \text{ И } (v_2 \text{ ИЛИ } v_3), \text{ где } \bar{v} \text{ означает «не } v\text{»}.$$

  Для выполнения формулы в КНФ нужно выполнить все дизъюнкции.

  Формулы в ДНФ состоят из дизъюнкций (операция ИЛИ) и конъюнкций (операция И). Предыдущую формулу можно записать в ДНФ таким образом:

  $$(\bar{v}_1 \text{ И } \bar{v}_2 \text{ И } v_3) \text{ ИЛИ } (\bar{v}_1 \text{ И } v_2 \text{ И } \bar{v}_3) \text{ ИЛИ } (\bar{v}_1 \text{ И } v_2 \text{ И } v_3) \text{ ИЛИ } (v_1 \text{ И } \bar{v}_2 \text{ И } v_2)$$

  Для выполнения формулы в ДНФ достаточно выполнить одну из конъюнкций.

  Решение задачи разрешимости формулы в ДНФ является тривиальным, поскольку любая формула ДНФ является выполнимой, если только *все конъюнкции не содержат* как литерал, так его дополнение (отрицание). Но задача выполнимости формулы в КНФ является NP-полной. Это кажется парадоксальным, поскольку посредством законов де Моргана можно преобразовать формулу из КНФ в ДНФ и наоборот. Причина заключается в том, что для такого преобразования может потребоваться экспоненциальное количество членов, вследствие чего само преобразование не может быть выполнено за полиномиальное время.

- *Каков размер дизъюнкций?*

  Задача $k$-SAT представляет собой частный случай задачи выполнимости, в которой каждая дизъюнкция состоит из самое большее $k$ переменных. При этом задача 1-SAT является тривиальной, поскольку для ее решения достаточно, чтобы истинное значение имела каждая переменная в любой дизъюнкции. Задача 2-SAT не является такой тривиальной, но все же ее можно решить за линейное время. Этот факт представляет интерес, поскольку при определенной изобретательности удается моделировать некоторые задачи в виде задачи 2-SAT. А вот задачи с дизъюнкциями, содержащими три переменные (т. е. 3-SAT), являются NP-полными.

- *Достаточно ли этого, чтобы удовлетворить большинству дизъюнкций?*

  Если требуется точное решение SAT-задачи, то вам в любом случае придется использовать какой-нибудь алгоритм перебора с возвратом — например, метод Дэвиса — Патнэма (Davis — Putnam). В наихудшем случае потребуется проверить $2^m$ наборов значений истинности, но, к счастью, существует много способов сократить пространство поиска. Хотя теоретически задача выполнимости является NP-полной, трудность конкретной задачи зависит от конкретного экземпляра задачи. Естественно определенные «случайные» экземпляры часто решаются с неожиданной легкостью, однако задача генерирования по-настоящему сложных экземпляров является нетривиальной.

Тем не менее иногда может быть полезно сделать задачу менее строгой, поставив в качестве цели выполнимость наибольшего количества дизъюнкций. Качество решений, полученных случайным методом или с помощью эвристического алгоритма, можно улучшить за счет применения оптимизационных методов — таких как метод имитации отжига. В самом деле, при присвоении переменным любого случайного набора значений истинности вероятность выполнимости каждой $k$-SAT дизъюнкции составляет $1 - (l/2)^k$, так что, скорее всего, с первой попытки выполнится большинство дизъюнкций. Однако завершить работу будет труднее. Задача поиска набора значений истинности, удовлетворяющего наибольшему количеству дизъюнкций, является NP-полной даже для экземпляров, не имеющих решения.

Когда мы сталкиваемся с задачей неизвестной сложности, доказательство ее NP-полноты может оказаться важным первым шагом в ее решении. Если вы думаете, что ваша задача может быть сложной, посмотрите, нет ли ее в списке сложных задач в книге [GJ79]. Если нет, то я рекомендую попробовать самостоятельно доказать сложность задачи, используя базовые задачи 3-SAT, вершинного покрытия, независимого множества, разбиения целого числа, клики и гамильтонова цикла. Стратегии доказательства сложности рассматриваются в *главе 11*.

### Реализации

В последние годы наблюдался заметный рост производительности средств решения задач выполнимости. На ежегодном соревновании по решению задач SAT определяются лучшие средства решения для нескольких категорий экземпляров задач. Исходный код всех этих решателей и другая информация доступны на веб-странице **http://www.satcompetition.org**.

Веб-сайт SAT Live! (**http://www.satlive.org**) — источник самых свежих научных работ, программ и тестовых наборов для задач выполнимости и родственных задач оптимизации логических функций.

#### *Примечания*

Наиболее полный обзор тестирования выполнимости на практике представлен в книге [KSBD07]. Алгоритм обхода с возвратом DPLL (Davis-Putnam-Logemann-Loveland) для решения задач выполнимости был опубликован в 1962 году. Методы локального поиска работают лучше на определенном классе задач, трудных для решателей DPLL. Обозрения в области тестирования выполнимости см. в [BHvM09], [GKSS08] и [KS07].

Алгоритмам выполнимости посвящена первая половина второй части Тома 4 работ Дональда Кнута, которая была издана отдельным выпуском (см. [Knu15]). В ней демонстрируется множество разнообразных применений задач выполнимости (включая игру в жизнь), а также содержится тщательное рассмотрение использования процедур поиска с возвратом для их решений.

В работе [DGH+02] описывается алгоритм для решения задачи 3-SAT за время $O^*(1{,}4802^n)$ в наихудшем случае. Обзор эффективных (но имеющих неполиномиальное время исполнения) алгоритмов решения NP-полных задач приводится в работе [Woe03].

Основным справочником по NP-полноте является книга [GJ79], содержащая список около 400 NP-полных задач. Уже долгие годы эта книга остается чрезвычайно полезным справочником. Я обращаюсь к ней чаще, чем к какой-либо другой. Обсуждению теоремы Кука

(см. [Coo71]), доказывающей сложность задачи выполнимости, посвящены работы [CLRS09], [GJ79] и [KT06]. Важность результатов работы Кука становится ясной из доклада Карпа (Karp) (см. [Kar72]), в котором доказывается сложность свыше 20 разных комбинаторных задач.

В работе [APT79] представлен алгоритм решения задач 2-SAT за линейное время. Интересное применение задачи 2-SAT для нанесения обозначений на карты излагается в работе [WW95]. Самый лучший известный эвристический алгоритм выдает приблизительное решение задачи MAX-2-SAT с точностью до 1,0741 (см. [FG95]).

### Родственные задачи

Условная оптимизация (см. *разд. 16.5*), задача коммивояжера (см. *разд. 19.4*).

# Задачи на графах с полиномиальным временем исполнения

Алгоритмические задачи на графах составляют приблизительно треть всего материала этой части книги. Более того, многие задачи из других разделов можно было бы сформулировать с тем же успехом на основе графов — например, задачу минимизации ширины ленты и задачу оптимизации конечного автомата. Одним из основных навыков хорошего алгоритмиста является способность определить правильное название той или иной задачи теории графов. Если вы знаете название своей задачи, вы найдете здесь точные инструкции, как приступить к ее решению.

В этой главе рассматриваются задачи, для решения которых существуют эффективные алгоритмы, время исполнения которых повышается полиномиально с увеличением размера графа. Так как часто существует несколько способов моделирования приложения, то имеет смысл, прежде чем использовать одну из более трудных постановок решаемой задачи, поискать в этой главе способ ее моделирования.

Графы легче всего воспринимать, когда они представлены в виде рисунков. Многие интересные свойства графов, например планарных, вытекают из типа рисунка графа. Поэтому мы также рассмотрим алгоритмы рисования графов, деревьев и планарных графов.

Многие сложные алгоритмы на графах трудно программировать. Но существуют уже готовые реализации — нужно только знать, где их можно найти. Лучшие источники реализаций алгоритмов на графах: библиотека LEDA (см. [MN99]) и библиотека Boost Graph Library (см. [SLL02]). Но для многих задач созданы и специальные программы.

Самые свежие обзоры разнообразных алгоритмов для работы с графами приведены в следующих источниках: [AB17], [TABN16] и [vL90a]. Среди представляющих интерес книг можно также порекомендовать:

- [Sed11] — том этого учебника, посвященный алгоритмам для работы с графами, содержит всеобъемлющее, но доступное введение в эту область;
- [AMO93] — хотя эта книга посвящена в основном потокам в сетях, в ней рассматривается целый набор алгоритмов, причем особое внимание уделяется исследованию операций;
- [Eve11] — уже не новый, но от этого не менее ценный учебник по алгоритмам для работы с графами, содержащий подробное обсуждение алгоритмов для проверки планарности графов.

# 18.1. Компоненты связности

**Вход.** Ориентированный или неориентированный граф $G$.

**Задача.** Разбить граф $G$ на части, или компоненты, так, чтобы вершины $x$ и $y$ принадлежали разным компонентам, если из $x$ в $y$ нет пути (рис. 18.1).

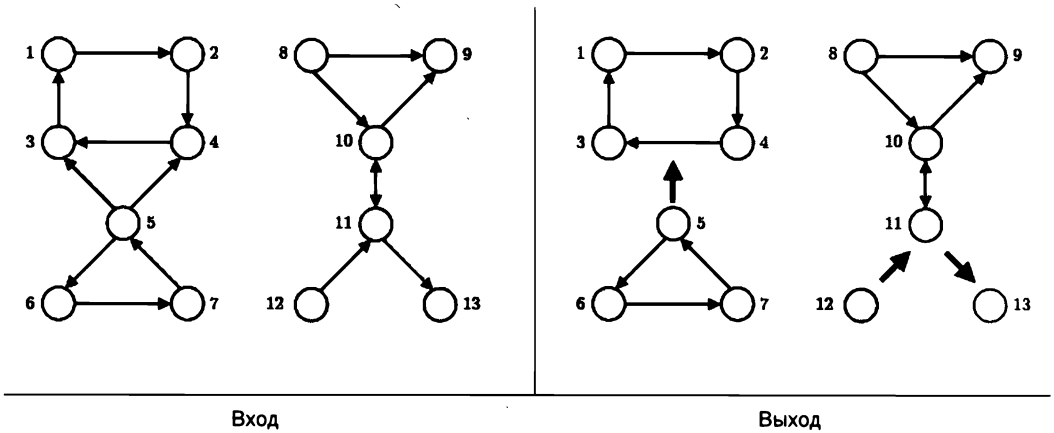



**Рис. 18.1.** Компоненты связности

### Обсуждение

Компоненты связности графа представляют собой отдельные части графа. Две вершины находятся в одном и том же компоненте графа $G$ тогда и только тогда, когда между ними существует какой-либо путь.

Задача поиска компонентов связности является центральной во многих приложениях на графах. Рассмотрим, например, задачу определения естественных кластеров в наборе элементов. Для ее решения мы представляем каждый элемент в виде вершины и соединяем ребром каждую пару элементов, которые считаются «подобными». Компоненты связности этого графа соответствуют разным классам элементов.

Проверка графа на наличие в нем компонентов связности является важным шагом предварительной обработки. При исполнении алгоритма только на одном компоненте несвязного графа часто удается выявить скрытые, труднообнаруживаемые ошибки. Проверка на связность выполняется так легко и быстро, что следует всегда проводить ее для графа входа, даже если вы уверены, что он просто *должен* быть связным.

Связность любого неориентированного графа можно проверить посредством обхода графа в глубину или в ширину. В действительности, не так важно, какой из этих методов обхода вы выберете. При любом обходе поле «номер компонента» для каждой вершины инициализируется нулевым значением, а потом начинается поиск первого компонента из вершины $v_1$. При открытии каждой вершины этому полю присваивается значение номера текущего компонента. По завершении первоначального обхода номер компонента увеличивается на единицу и поиск выполняется снова, начиная с первой вершины, у которой номер компонента все еще равен нулю. При правильной реализа-

ции с использованием списков смежности (как показано в *разд. 7.7.1*) этот алгоритм имеет время исполнения $O(n + m)$.

На практике также возникают другие вопросы, касающиеся связности.

♦ *Является ли граф ориентированным?*

Для ориентированных графов существуют два понятия компонентов связности:

- ориентированный граф является *сильно связным* (strongly connected), если между любой парой его вершин существует ориентированный путь;
- ориентированный граф является *слабо связным* (weakly connected), если при игнорировании ориентации ребер он окажется связным.

Различие между ними легко понять при рассмотрении сети улиц с односторонним и двусторонним движением. Сеть является сильно связной, если возможно проехать между любыми двумя точками, не нарушая правил. Если между любыми двумя точками можно проехать как не нарушая, *так и* нарушая правила, то сеть является слабо связной. Если же попасть из какой-либо точки $A$ в какую-либо точку $B$ никак нельзя, то сеть является несвязной.

Слабо и сильно связные компоненты определяют однозначные разбиения вершин. На рис. 18.1 представлен ориентированный граф, состоящий из двух слабо (*слева*) или пяти сильно (*справа*) связных компонентов (также называемых *блоками* графа $G$).

Проверку ориентированного графа на слабую связность можно с легкостью провести за линейное время. Просто сделаем все ребра графа неориентированными и выполним на нем алгоритм поиска компонентов связности на основе обхода в глубину. Проверку на сильную связность провести сложнее. Самый простой алгоритм с линейным временем исполнения осуществляет поиск обходом в глубину, начиная с любой вершины $v$, чтобы продемонстрировать достижимость всех вершин графа из этой вершины. Затем мы создаем транспозицию $G'$, поменяв направление всех ребер графа $G$ на обратное. Чтобы определить, достижимы ли все вершины графа $G$ из вершины $v$, достаточно выполнить обход графа $G'$ из этой вершины. Граф $G$ является сильно связным тогда и только тогда, когда вершина $v$ достижима из любой вершины в графе $G$ и все вершины в графе $G$ достижимы из вершины $v$.

Все сильно связные компоненты графа $G$ можно извлечь за линейное время, используя более сложные алгоритмы типа обхода в глубину. Обобщение только что описанной идеи двойного обхода в глубину с легкостью поддается программированию, но понять, как оно работает, несколько сложнее.

1. Выполняем обход в глубину, начав с произвольной вершины графа $G$ и нумеруя вершины в порядке завершения их обработки (а не открытия).
2. Изменяем направление каждого ребра графа $G$, создав, таким образом, граф $G'$.
3. Выполняем обход в глубину графа $G'$, начиная с вершины графа $G$, имеющей наибольший номер. Если не удается обойти граф $G'$ полностью, выполняем новый обход, начиная с еще не посещенной вершины с наибольшим номером.
4. Каждое созданное на шаге 3 дерево обхода в глубину определяет сильно связный компонент.

В *разд. 7.10.2* приводится моя реализация этого двухпроходного алгоритма. В любом случае проще использовать готовую реализацию, чем изучать описание алгоритма в учебнике.

- *Какое самое слабое место в моем графе/сети?*

  Прочность цепочки измеряется прочностью ее самого слабого звена. Потеря одного или нескольких внутренних звеньев разорвет цепочку на части. *Связность* графа измеряет его «прочность» — количество ребер или вершин, которое нужно удалить, чтобы разбить граф на части. Связность является важным понятием в проектировании сетей и других задачах структурирования.

  Алгоритмические задачи связности графов рассматриваются в *разд. 18.8*. В частности, *двусвязные компоненты* представляют собой части графа, получаемые в результате разрыва ребер, имеющих общую вершину. Все компоненты двусвязности можно найти за линейное время посредством обхода в глубину. Реализация этого алгоритма представлена в *разд. 7.9.2*. Вершины, удаление которых нарушает связность графа, принадлежат нескольким компонентам двусвязности, ребра которых однозначно распределяются между компонентами.

- *Является ли граф деревом? Как найти цикл, если он существует?*

  Задача поиска цикла возникает часто, особенно с ориентированными графами. Например, проверка, возможна ли взаимная блокировка последовательности условий, часто сводится к задаче выявления цикла. Если я жду Фреда, Фред ждет Мэри, а Мэри ждет меня, то налицо цикл и взаимная блокировка.

  Для неориентированных графов аналогичной задачей является идентификация дерева. По определению дерево является неориентированным связным бесконтурным графом. Для проверки графа на связность можно использовать обход в глубину. Если граф является связным и имеет $n - 1$ ребер для $n$ вершин, то этот граф является деревом.

  Обход в глубину можно использовать для поиска циклов как в ориентированных, так и в неориентированных графах. Всякий раз, когда в процессе обхода в глубину мы обнаруживаем обратное ребро (т. е. ребро к вершине-предшественнику в дереве обхода в глубину), это обратное ребро и дерево совместно определяют ориентированный цикл. Ориентированные графы, не содержащие контуров (циклов), называются *бесконтурными*. Фундаментальной операцией на бесконтурных ориентированных графах является топологическая сортировка (см. *разд. 18.2*).

### Реализации

Все реализации структур данных для работы с графами, описанные в *разд. 15.4*, включают реализацию обхода в ширину и обхода в глубину и, соответственно, определенную степень проверки на связность. Библиотека Boost Graph Library для C++ (см. [SLL02]) предоставляет реализации компонентов связности. Ее можно загрузить с веб-сайта **http://www.boost.org/libs/graph/doc**. Библиотека LEDA (см. *разд. 22.1.1*) содержит реализации на языке C++ этих компонентов, а также двусвязных и трехсвязных компонентов и обходов в глубину и ширину.

Для программистов, пишущих на языке Java, библиотека JUNG (**http://jung.sourceforge.net/**) содержит реализации алгоритмов компонентов двусвязности, а библиотека JGraphT ((**https://jgrapht.org/**) — компоненты сильной двусвязности.

На мой взгляд, лучшей реализацией всех основных алгоритмов проверки на связность (включая компоненты сильной связности и компоненты двусвязности) на языке C является библиотека, разработанная для этой книги. Подробности вы найдете в *разд. 22.1.9*.

#### Примечания

Обход в глубину впервые был использован в XIX столетии для поиска выхода из лабиринтов (см. [Luc91] и [Tar95]). А обход в ширину — в 1957 году для поиска кратчайшего пути (см. [Moo59]).

В работах [HT73b] и [Tar72] установлено, что обход в глубину является фундаментальным методом построения эффективных алгоритмов на графах. Алгоритмы обхода в глубину и обхода в ширину приводятся в каждой книге, посвященной алгоритмам, причем книга [CLRS09] содержит, пожалуй, наиболее полное описание этих алгоритмов.

Первый алгоритм для поиска компонентов сильной связности предложен в работе [Tar72], а его описание приведено в книгах [BvG99], [Eve11] и [Man89]. Еще один алгоритм для поиска компонентов сильной связности, более легкий для программирования и более изящный, разработан Шариром (Sharir) и Косараю (Kosaraju). Хорошие описания этого алгоритма приведены в книгах [AHU83] и [CLRS09]. В своей работе [CM96] Чериян (Cheriyan) и Мэлхорн (Mehlhorn) представляют улучшенные алгоритмы для некоторых задач на плотных графах, включая поиск компонентов сильной связности.

Поиск обходом в глубину трудно реализовать параллельным способом. Параллельные алгоритмы для связных компонентов по модели MapReduce и другим моделям вычислений представляются в работах [KLM+14] и [SRM14].

Случайные графы проявляют интересные свойства связности. В частности, как только количество ребер в графе превышает определенное (на удивление низкое) пороговое значение, такой граф, скорее всего, содержит один громадный связный компонент и небольшое количество крошечных компонентов. Например, случайный граф, содержащий всего лишь $n \ln 2 = 0{,}693n$ ребер, наверняка содержит связный компонент с $n/2$ вершинами. Аналогично, любая крупная социальная сеть (наподобие Facebook) предположительно содержит один крупный связный компонент, включающий почти всех ее членов, за исключением принципиальных одиночек и новых пользователей. Это явление составляет большую часть основания, на котором зиждется гипотеза «шести шагов», утверждающая, что любых двух людей в мире связывает цепочка из самое большее шести звеньев (т. е. промежуточных людей). Предмет возникновения крупного связного компонента рассматривается в каждом учебнике по теории случайных графов, включая [Bol01] и [JLR00].

#### Родственные задачи

Связность ребер и вершин (см. *разд. 18.8*), поиск кратчайшего пути (см. *разд. 18.4*).

## 18.2. Топологическая сортировка

**Вход.** Бесконтурный (ациклический) ориентированный граф $G = (V, E)$, также называемый *частично упорядоченным множеством* (partially ordered set).

**Задача.** Найти линейное упорядочение вершин $V$ такое, что для каждого ребра $(i, j)$ вершина $i$ находится слева от вершины $j$ (рис. 18.2).

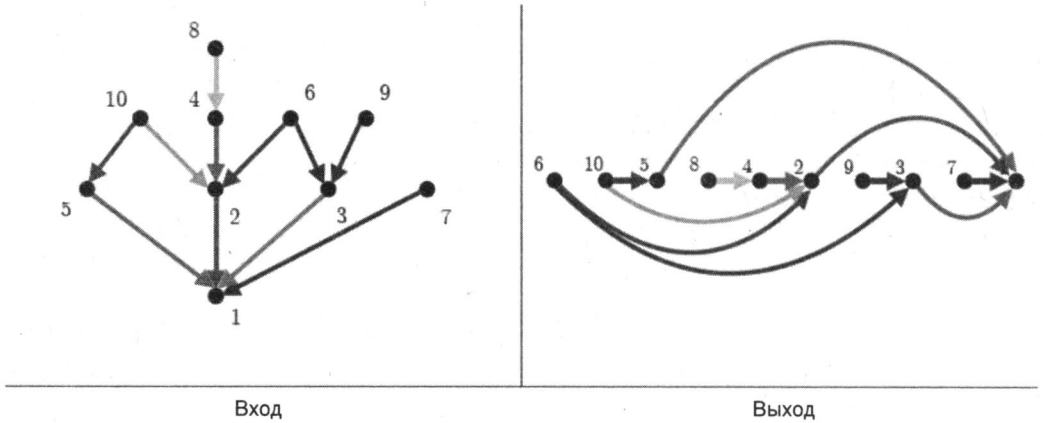



**Рис. 18.2.** Топологическая сортировка

### Обсуждение

Подзадача топологической сортировки возникает в большинстве алгоритмов на ориентированных ациклических графах. Топологическая сортировка упорядочивает вершины и ребра бесконтурного ориентированного графа простым и непротиворечивым образом и поэтому играет ту же самую роль для бесконтурных ориентированных графов, что и обход в глубину для общих графов.

Топологическую сортировку можно использовать для создания расписания работ с ограничивающими условиями очередности. Если у нас имеется набор задач, которые должны выполняться в определенной последовательности, то эти ограничивающие условия очередности формируют бесконтурный ориентированный граф, и любая топологическая сортировка на нем, иногда называемая *линейным расширением* (linear extension), определяет такой порядок выполнения этих работ, при котором каждая из них выполняется только после удовлетворения ее ограничивающих условий.

В отношении топологической сортировки справедливы следующие три важные утверждения:

- топологическую сортировку можно выполнять *только* на бесконтурных ориентированных графах, поскольку любой ориентированный цикл принципиально противоречит линейному порядку выполнения работ;
- топологическую сортировку можно выполнить на *любом* бесконтурном ориентированном графе, поэтому для какого бы то ни было разумного набора ограничивающих условий очередности работ существует, по крайней мере, одно расписание их выполнения;
- для бесконтурного ориентированного графа обычно можно выполнить несколько разных топологических упорядочений — особенно при небольшом количестве ограничивающих условий. Так, для $n$ работ, не имеющих никаких ограничивающих условий, все $n!$ перестановок работ являются допустимыми топологическими упорядочениями.

Самый концептуально простой алгоритм топологической сортировки выполняет обход в глубину бесконтурного ориентированного графа, чтобы найти полный набор вершин-

истоков, т. е. вершин, у которых количество входящих дуг равно нулю. Любой бесконтурный ориентированный граф должен иметь, по крайней мере, одну такую вершину-исток. Вершины-истоки могут находиться в начале любого расписания, не нарушая никаких ограничивающих условий. Удаление всех исходящих ребер этих вершин-истоков создаст новые вершины-истоки, которые будут располагаться справа от первого набора. Эта процедура повторяется до тех пора, пока не будут учтены все вершины. При разумном выборе структур данных (списков смежности и очередей) этот алгоритм будет иметь время исполнения $O(n + m)$.

Альтернативный ему алгоритм основан на том обстоятельстве, что упорядочение всех вершин по времени завершения обхода в глубину в убывающем порядке дает линейное расширение. Реализация этого алгоритма с доказательством его правильности приводится в *разд. 7.10.1*.

В топологической сортировке следует принимать во внимание ответы на следующие два вопроса:

- *как получить все линейные расширения?*

  Иногда нужно создать *все* линейные расширения бесконтурного ориентированного графа, чтобы, скажем, идентифицировать наилучшее расписание согласно вторичному критерию, которое удовлетворяет всем ограничениям на предшествующие вершины. В таких случаях вы должны отдавать себе отчет в том, что количество линейных расширений может возрастать экспоненциально по отношению к размеру бесконтурного ориентированного графа. Даже сама задача подсчета количества всех линейных расширений является NP-полной.

  Алгоритмы для перечисления всех линейных расширений бесконтурного ориентированного графа основаны на обходе с возвратом. В них создаются все возможные слева направо упорядочения, где кандидатами для следующей вершины являются все вершины с нулевой степенью захода. Прежде чем продолжать обход, из выбранной вершины удаляются все исходящие ребра. Оптимальный алгоритм для перечисления линейных расширений рассматривается далее.

  Алгоритмы для создания случайных линейных расширений начинают с произвольного линейного расширения. В нем последовательно выбираются пары вершин, которые меняются местами, если получающаяся в результате такого обмена перестановка остается топологической сортировкой. При выборе достаточного количества случайных пар вершин этот процесс равномерно выбирает линейное расширение. Подробности см. в *подразделе «Примечания»*.

- *как поступать, когда граф не является бесконтурным?*

  Когда набор ограничивающих условий содержит внутренние противоречия, естественно возникает задача поиска наименьшего набора условий, удаление которого устраняет все конфликты. Наборы непрестижных работ (вершин) или ограничений (ребер), после удаления которых ориентированный граф становится бесконтурным, называются *разрывающими множествами вершин* (feedback vertex set) и *разрывающими множествами дуг* (feedback arc set) соответственно. Эти структуры рассматриваются в *разд. 19.11*. К сожалению, задачи поиска обоих наборов являются NP-полными.

Так как алгоритм топологической сортировки на основании обхода в глубину зацикливается, как только он находит вершину в ориентированном контуре, мы можем удалить проблемное ребро или вершину и продолжать работу. В конечном итоге этот простой и эффективный эвристический подход делает ориентированный граф бесконтурным, но может удалить больше элементов, чем необходимо. Аппроксимирующий алгоритм для решения этой задачи описывается в *разд. 12.4.2*.

### Реализации

По сути, все упоминаемые в *разд. 15.4* реализации структур данных для представления графов, включая библиотеку Boost Graph Library (**http://www.boost.org/libs/graph/doc**) и библиотеку LEDA (см. *раздел 22.1.1*), содержат реализации топологической сортировки. Для языка Java советую обратить внимание на библиотеку JGraphT (**https://jgrapht.org/**).

Сервер комбинаторных объектов на веб-сайте **http://combos.org/** предоставляет программы на языке C для генерирования линейных расширений как в лексикографическом порядке, так и в порядке кода Грея, а также средства для их перечисления. Сервер оснащен интерактивным интерфейсом.

На мой взгляд, лучшей реализацией всех основных алгоритмов решения задач на графах (включая топологическую сортировку), на языке C, является библиотека, разработанная для этой книги. Подробности см. в *разд. 22.1.9*.

#### Примечания

Хорошие описания топологической сортировки включают книги [CLRS09] и [Man89]. Для сортировки графов на внешних устройствах хранения не известно алгоритмов с доказуемо эффективным вводом/выводом. Впрочем, Аджуани (Ajwani) в [ACLZ11] приводит отчет о результатах своей работы по разработке топологической сортировки для массивных графов. В своей работе [BW91] Брайтвэл (Brightwell) и Уинклер (Winkler) доказали, что задача подсчета количества линейных расширений частичного порядка является #P-полной даже для частично упорядоченных множеств высотой, равной 2 (см. [DP18]). Класс сложности #P включает класс сложности NP, следовательно, любая #P-полная задача должна быть NP-сложной.

Прусс (Pruesse) и Раски (Ruskey) в своей работе ([PR86]) приводят алгоритм для генерирования линейных расширений бесконтурного ориентированного графа за постоянное амортизированное время. Кроме этого, каждое расширение отличается от своего предшественника перестановкой только одного или двух смежных элементов. Этот алгоритм можно использовать для подсчета линейных расширений $e(G)$ $n$-вершинного графа $G$ за время $O(n^2 + e(G))$. Для перечисления линейных расширений также можно использовать метод поиска в обратном направлении Ависа (Avis) и Фукуды (Fukuda), представленный в работе [AF96]. Программа обхода с возвратом для генерирования всех линейных перечислений описана в работе [KS74] Дональда Кнута.

В работе [Hub06] Губера (Huber) представлен алгоритм случайной выборки с равномерным распределением линейных расширений из произвольного частичного порядка за время $O(n^3 \lg n)$, что улучшает результаты работы алгоритма, представленного в работе [BD99].

### Родственные задачи

Сортировка (см. *разд. 17.1*), разрывающее множество вершин и ребер (см. *разд. 19.11*).

# 18.3. Минимальные остовные деревья

**Вход.** Граф $G = (V, E)$ со взвешенными ребрами.

**Задача.** Найти подмножество ребер $E' \subset E$, которое определяет дерево на вершинах $V$ и имеет минимальный вес (рис. 18.3).

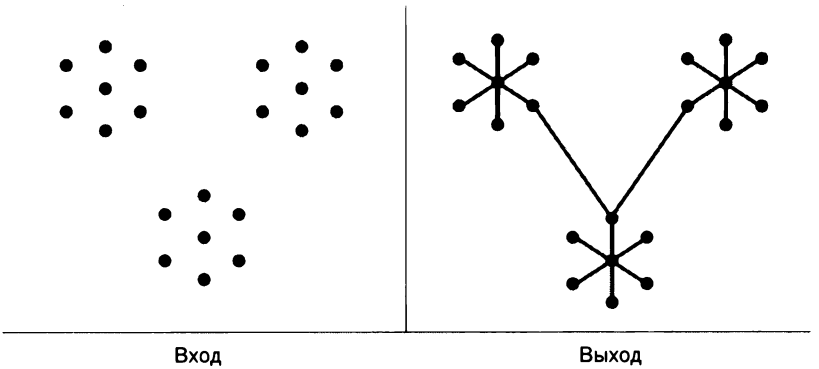



**Рис. 18.3.** Поиск минимального остовного дерева

### Обсуждение

Минимальное остовное дерево графа определяет подмножество ребер с минимальным весом, которое объединяет граф в один компонент связности. В решении задачи построения минимального остовного дерева могут быть заинтересованы, например, телефонные компании, поскольку минимальное остовное дерево набора объектов определяет схему соединения этих объектов в сеть с использованием кабеля наименьшей длины. Таким образом, задача построения минимального остовного дерева является основной задачей проектирования сетей.

Важность минимальных остовных деревьев объясняется несколькими причинами:

- они легко и быстро реализуются на компьютере и образуют разреженные подграфы, которые содержат много информации о первоначальных графах;
- они предоставляют способ определить кластеры в наборах точек. Удаление всех длинных ребер из минимального остовного дерева оставляет компоненты связности, которые определяют естественные кластеры в наборе данных. Например, после удаления двух длинных ребер в выходном графе на рис. 18.3 (*справа*) остаются три естественных кластера;
- с их помощью можно получить приблизительные решения сложных задач — таких как задача построения дерева Штейнера и задача коммивояжера;
- они могут быть использованы в учебных целях, поскольку алгоритмы для построения минимального остовного дерева дают графическую иллюстрацию того, как «жадные» алгоритмы иногда выдают доказуемо оптимальные решения.

Известны три классических алгоритма для эффективного создания минимальных остовных деревьев. Подробная реализация двух из них (алгоритма Прима и алгоритма

Крускала) совместно с доказательством их правильности дается в *разд. 8.1*. Третий алгоритм (Борувки) менее известен, хотя он был изобретен первым, прост в реализации и более эффективен.

Далее даются краткие описания и псевдокоды этих алгоритмов.

- *Алгоритм Крускала.*

  В начале каждая вершина представляет отдельное дерево, и эти деревья сливаются в одно путем последовательного добавления ребер с наименьшим весом, которые не создают цикл. Псевдокод этого алгоритма представлен в листинге 18.1.

**Листинг 18.1. Алгоритм Крускала**

```
Kruskal(G)
   Сортируем ребра по весу в возрастающем порядке
   count = 0
   while (count < n - 1) do
      get next edge (v, w)
      if (component (v) ≠ component(w))
         add to T
         component(v) = component(w)
         count++
```

  Проверку, какой компонент выбирать, можно эффективно реализовать с помощью структуры данных «объединение-поиск» (см. *разд. 15.5*), что даст нам алгоритм с временем исполнения $O(m \lg m)$.

- *Алгоритм Прима.*

  Начинает работу с произвольной вершины $v$ и «выращивает» дерево из этой вершины, в цикле выбирая ребро с наименьшим весом, которое присоединяет к дереву какую-либо новую вершину. В процессе исполнения каждая вершина помечается или как входящая в дерево, или как открытая, но еще не добавленная в дерево, или как не увиденная (т. е. расположенная на расстоянии более одного ребра от дерева). Псевдокод этого алгоритма представлен в листинге 18.2.

**Листинг 18.2. Алгоритм Прима**

```
Prim(G)
   Выбираем произвольную вершину, с которой надо начинать построение дерева
   while (имеются вершины, не включенные в дерево)
      выбираем ребро минимального веса между деревом и вершиной вне дерева
      добавляем выбранное ребро и вершину в дерево
      обновляем стоимость для всех затрагиваемых ребер вне дерева
```

  Этот алгоритм создает остовное дерево для любого связного графа, поскольку циклы не могут быть внесены, добавлением ребер между вершинами в дереве и вне его. То, что это дерево действительно имеет минимальный вес, доказывается методом «от противного». Используя простые структуры данных, можно реализовать алгоритм Прима с временем исполнения $O(n^2)$.

- *Алгоритм Борувки.*

  Основан на том обстоятельстве, что в минимальном остовном дереве должны быть инцидентные каждой вершине ребра наименьшего веса. Результатом объединения этих ребер будет остовной лес, содержащий как минимум $n/2$ деревьев. Теперь для каждого из этих деревьев $T$ выбираем такое ребро $(x, y)$ наименьшего веса, для которого $x \in T$ и $y \notin T$. Все эти ребра опять должны быть в минимальном остовном дереве, а результатом их объединения снова будет остовной лес, содержащий самое большее половину предыдущего количества деревьев. Псевдокод этого алгоритма представлен в листинге 18.3.

**Листинг 18.3. Алгоритм Борувки**

```
Boruvka(G)
   Инициализируем остовной лес F, состоящий из n одновершинных деревьев
      while (лес F содержит больше чем одно дерево)
          for each (дерева T в лесу F) находим ребро с наименьшим весом
                                                          от T к G – T
          добавляем все выбранные ребра в лес F,
          сливая вместе пары деревьев
```

  В каждой итерации количество деревьев уменьшается, по крайней мере, вдвое, что дает нам минимальное остовное дерево после самое большее $\log n$ итераций, каждая из которых выполняется за линейное время. В целом мы получаем алгоритм с временем исполнения $O(m \log n)$, причем без использования каких-либо хитроумных структур данных.

Построение минимального остовного дерева является только одной из нескольких задач об остовном дереве, возникающих на практике. Чтобы разобраться в них, попытайтесь найти ответы на следующие вопросы.

- *Одинаковый ли вес у всех ребер графа?*

  Любое остовное дерево из $n$ вершин имеет ровно $n - 1$ ребер. Таким образом, если граф невзвешенный, то *любое* остовное дерево должно быть минимальным остовным деревом. Корневое остовное дерево можно найти за линейное время посредством обхода в глубину или в ширину. Обход в глубину обычно выдает высокие и тонкие деревья, а обход в ширину отражает структуру расстояний графа.

- *Какой алгоритм использовать: Прима или Крускала?*

  В том виде, в каком они реализованы в *разд. 8.1*, алгоритм Прима имеет время исполнения $O(n^2)$, а алгоритм Крускала — $O(m \log m)$. Таким образом, алгоритм Прима эффективен на плотных графах, а алгоритм Крускала — на разреженных. Вместе с тем, используя более сложные структуры данных, можно создать реализации алгоритма Прима с временем исполнения $O(m + n \lg n)$, а реализация алгоритма Прима с использованием парных пирамид будет самой быстрой на практике как для разреженных, так и для плотных графов.

- *Что предпринять, если входные данные — не граф, а точки на плоскости?*

  Геометрические экземпляры, состоящие из $n$ точек в $d$ измерениях, можно решать, создав граф полного расстояния за время $O(n^2)$, а затем построив минимальное

остовное дерево этого графа. Но для точек на плоскости более эффективным подходом оказывается решение непосредственно геометрической версии задачи. Сначала создадим триангуляцию Делоне (Delaunay triangulation) для этих точек (см. *разд. 20.3* и *20.4*), в результате чего получим граф с $O(n)$ ребрами, который содержит все ребра минимального остовного дерева нашего множества точек. Алгоритм Крускала находит минимальное остовное дерево в этом разреженном графе за время $O(n \lg n)$.

- *Как можно найти остовное дерево, не имеющее вершин с высокой степенью?*

  Другой распространенной целью задач об остовном дереве является минимизация максимальной степени вершин, что обычно нужно для уменьшения до минимума исходящих разветвлений в сети. К сожалению, задача построения остовного дерева с максимальной степенью вершин, равной 2, является NP-полной задачей, поскольку она идентична задаче поиска гамильтонова пути. Но существуют эффективные алгоритмы для создания остовных деревьев с максимальной степенью вершин на единицу больше, чем требуемая, чего должно быть достаточно на практике.

### Реализации

Среди реализаций структур данных для представления графов из *разд. 15.4* обязательно должны присутствовать реализации алгоритма Прима и/или алгоритма Крускала. Этому требованию отвечают библиотеки Boost Graph Library (см. [SLL02]) (**http://www.boost.org/libs/graph/doc**) и LEDA (см. *разд. 22.1.1*). Веб-сайт JGraphT (**https://jgrapht.org**) содержит обширную библиотеку алгоритмов для работы с графами на языке Java, включающую алгоритмы Борувки, Крускала, Прима и другие варианты.

Эксперименты с измерением времени исполнения алгоритмов для построения минимального остовного дерева показывают противоречивые результаты, дающие основание полагать, что разница в их производительности слишком незначительная. Реализации алгоритмов Прима, Крускала и Черитона — Тарьяна (Cheriton — Tarjan) на языке Pascal описываются в книге [MS91]. Здесь же приводится подробный анализ экспериментальных данных, доказывающий, что на большинстве графов самым быстрым будет алгоритм Прима, реализованный с правильно построенной очередью с приоритетами. В программе Standford GraphBase самым быстрым из четырех реализаций разных алгоритмов построения минимального остовного дерева оказался алгоритм Крускала (см. *разд. 22.1.7*).

Библиотека Combinatorica (см. [PS03]) содержит реализацию (на языке пакета Mathematica) алгоритма Крускала для построения минимального остовного дерева и быстрого подсчета количества остовных деревьев в графе. Подробности см. в *разд. 22.1.8*.

На мой взгляд, лучшей реализацией на языке C всех основных алгоритмов на графах, включая алгоритмы для построения минимальных остовных деревьев, является библиотека, разработанная для этой книги. Подробности вы найдете в *разд. 22.1.9*.

#### *Примечания*

Алгоритм Борувки для решения задачи построения минимального остовного дерева был разработан в 1926 году, т. е. намного раньше, чем алгоритмы Прима (см. [Pri57]) и Крускала, которые появились только в середине 1950-х годов. Алгоритм Прима был заново открыт Дейкстрой (см. [Dij59]). Дополнительную информацию об истории алгоритмов

построения минимального остовного дерева можно найти в [GH85]. By (Wu) и Чао (Chao) написали монографию [WC04] о задаче построения минимального остовного дерева и родственных задачах.

Самые быстрые реализации алгоритмов Прима и Крускала используют пирамиды Фибоначчи (см. [FT87]). Однако позже были предложены парные пирамиды, которые обеспечивают такую же производительность, но при меньших накладных расходах. Эксперименты с парными пирамидами описаны в журнале [SV87]. Эффективные параллельные алгоритмы для моделей типа MapReduce представлены в [ANOY14].

Алгоритм, получаемый в результате простой комбинации алгоритма Борувки с алгоритмом Прима, имеет время исполнения, равное $O(m \lg \lg n)$. Выполните $\lg \lg n$ итераций алгоритма Борувки, чтобы получить лес из самое большее $n / \lg n$ деревьев. Теперь создайте граф $G'$, в котором каждое дерево этого леса представлено одной вершиной, а вес ребра между деревьями $T_i$ и $T_j$ установите равным весу самого легкого ребра $(x, y)$, где $x \in T_i$, а $y \in T_j$. Минимальное остовное дерево графа $G'$ совместно с ребрами, выбранными алгоритмом Борувки, образует минимальное остовное дерево графа $G$. Время исполнения алгоритма Прима (реализованного с использованием пирамид Фибоначчи) на этом графе из $n / \lg n$ вершин и $m$ ребер составляет $O(m)$.

История выяснения оптимального времени построения минимальных остовных деревьев выглядит примерно так. В своей работе [KKT95] Каргер (Karger), Кляйн (Klein) и Тарьян (Tarjan) предложили рандомизированный алгоритм на основе алгоритма Борувки для построения минимального остовного дерева за линейное время. В свою очередь, Шазель (Chazelle) в работе [Cha00] представил алгоритм с временем исполнения $O(n\alpha(m, n))$, где $\alpha(m, n)$ является функцией, обратной функции Аккермана. Наконец, Петти (Pettie) и Рамачандран (Ramachandran) в работе [PR02] описали доказуемо оптимальный алгоритм, точное время исполнения которого неизвестно (как бы странно это ни звучало), но лежит в диапазоне между $\Omega(n + m)$ и $O(n\alpha(m, n))$.

*Остовным подграфом* $S(G)$ графа $G$ называется подграф, обеспечивающий эффективный компромисс между двумя противоречащими друг другу целями проектирования сетей. Говоря конкретно, общий вес *остовного подграфа* $S(G)$ должен быть близок к весу минимального остовного дерева полного графа $G$, в то же самое время гарантируя, что кратчайший путь между вершинами $x$ и $y$ в графе $S(G)$ близок к кратчайшему пути в полном графе $G$. Самый свежий исчерпывающий обзор достижений в этой области представлен в монографии [NS07].

Алгоритм для вычисления евклидова минимального остовного дерева разработан Шамосом (Shamos). Он обсуждается в учебниках по вычислительной геометрии — таких как [dBvKOS08] и [PS85].

В работе [FR94] Фурера (Furer) и Рагхавачари (Raghavachari) представлен алгоритм для создания остовного дерева почти минимальной степени — всего лишь на единицу больше, чем степень остовного дерева наименьшей степени. Эта ситуация аналогична вытекающей из теоремы Визинга (Vizing) для раскраски ребер, которая позволяет построить аппроксимирующий алгоритм, выдающий решение с аддитивным фактором, равным единице. В работе [SL07] представлен полиномиальный алгоритм построения остовного дерева с максимальной степенью, равной или меньшей чем $k + 1$, стоимость которого не больше стоимости оптимального минимального остовного дерева с максимальной степенью, равной или меньшей чем $k$.

Алгоритмы построения минимального остовного дерева можно рассматривать в виде *матроидов* (matroids), которые представляют собой системы подмножеств, замкнутые по включению. Максимальное взвешенное независимое множество матроида можно найти, используя «жадный» алгоритм. Связь между «жадными» алгоритмами и матроидами

установлена Эдмондсом (Edmonds) в его работе [Edm71]. Теория матроидов обсуждается в [GM12], [Law11] и [PS98].

Динамические алгоритмы на графах стремятся сохранять инвариант (например, минимальное остовное дерево) при операциях вставки или удаления ребер. В работе [HdIT01] представлен эффективный детерминистский алгоритм для поддержания минимальных остовных деревьев (и нескольких других инвариантов) за амортизированное полилогарифмическое время при каждом обновлении.

Алгоритмы генерирования остовных деревьев в порядке возрастания веса обсуждаются в работе [Gab77]. Полный набор остовных деревьев невзвешенного графа можно сгенерировать за постоянное амортизированное время. Обзор алгоритмов генерирования, ранжирования и остовных деревьев представлен в работе [Rus03].

### Родственные задачи

Дерево Штейнера (см. *разд. 19.10*), задача коммивояжера (см. *разд. 19.4*).

## 18.4. Поиск кратчайшего пути

**Вход.** Граф $G$ со взвешенными ребрами и две вершины: $s$ и $t$.

**Задача.** Найти кратчайший путь от вершины $s$ к вершине $t$ (рис. 18.4).

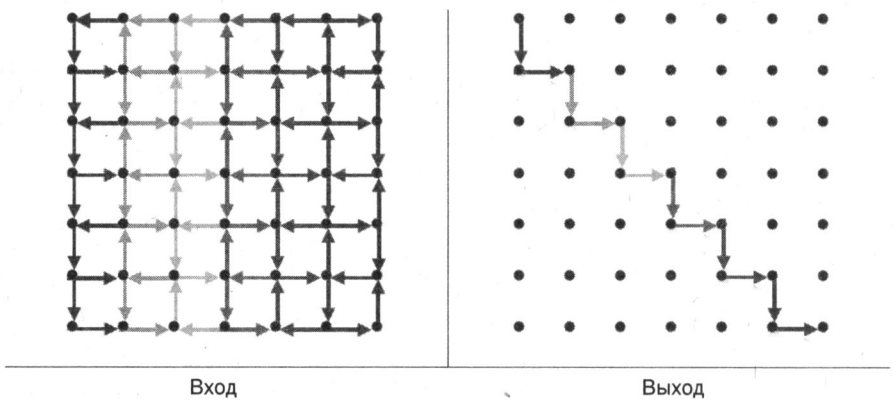



**Рис. 18.4.** Поиск кратчайшего пути

### Обсуждение

Задача поиска кратчайшего пути в графе имеет несколько применений, иногда довольно неожиданных:

- *транспортировка или связь.*

  Задача состоит в том, чтобы найти наилучший маршрут — например, для отправки грузовика с товаром из Чикаго в Феникс, или самый быстрый маршрут для направления сетевых пакетов к месту назначения;

- *сегментация изображения.*

  Разбиение оцифрованного изображения на области, содержащие отдельные объекты. Границы областей при этом допустимо рассматривать как пути по изображе-

нию, не проходящие через пикселы объекта, насколько это возможно. Сетку пикселов можно смоделировать в виде графа, стоимость ребер которого отражает изменение цвета между соседними пикселами. Кратчайший путь от точки $x$ к точке $y$ в таком взвешенном графе определяет потенциально успешную линию раздела;

- *системы распознавания речи.*

  Вспомним историю из жизни (см. *разд. 8.4*), в которой нужно было разработать определенные грамматические ограничивающие условия, чтобы из набора возможных слов выбрать правильное для каждой позиции в предложении. Такие задачи широко распространены в системах распознавания речи и обработки текстов на естественных языках. Для их решения создается граф, в котором вершины представляют возможные трактовки слова, а смежные трактовки соединяются ребрами. Если для каждого ребра установить вес, пропорциональный вероятности перехода, то кратчайший путь будет определять наилучшую трактовку предложения;

- *визуальное представление графа.*

  На рисунке «центр» графа должен находиться в центре страницы. В идеальном варианте это будет вершина с минимальным расстоянием до всех других вершин графа. Чтобы найти эту центральную вершину, нужно знать длину кратчайшего пути между всеми парами вершин.

Основным алгоритмом поиска кратчайшего пути является *алгоритм Дейкстры*, который эффективно вычисляет кратчайший путь от заданной начальной вершины $x$ ко всем $n-1$ другим вершинам. При каждой итерации алгоритм находит новую вершину $v$, для которой известен кратчайший путь из вершины $x$. Мы создаем набор вершин $S$, к которым имеется кратчайший путь из вершины $x$, и с каждой итерацией увеличиваем этот набор на одну вершину. При каждой итерации алгоритм находит ребро $(u, v)$, где $u \in S$ и $v \in V - S$ такие, что:

$$dist(x,y) = \min_{u \in S}(dist(x,u) + weight(u,v)),$$

где *dist* — расстояние; *weight* — вес; min *dist* — минимальное расстояние.

Это ребро $(u, v)$ добавляется к *дереву кратчайшего пути* с корнем в вершине $x$, которое описывает все кратчайшие пути из этой вершины.

В *разд. 8.3.1* приводится реализация алгоритма Дейкстры с временем исполнения $O(n^2)$. Как показано далее, можно получить лучшее время исполнения, используя более сложные структуры данных. Если нам достаточно найти кратчайший путь от $x$ к $y$, то выполнение алгоритма следует прервать, как только вершина $y$ окажется в множестве $S$.

Алгоритм Дейкстры применяется для поиска кратчайшего пути из одной начальной точки на графах с положительным весом ребер. Но иногда существуют ситуации, когда нужно воспользоваться другим алгоритмом. Для этого надо найти ответы на следующие вопросы.

- *Является ли граф взвешенным?*

  Если граф невзвешенный, то простой обход в ширину, начиная с вершины-источника, предоставит кратчайший путь ко всем другим вершинам за линейное время. Более сложный алгоритм требуется только тогда, когда ребра имеют разный вес.

- *Есть ли в графе ребра с отрицательным весом?*

  Алгоритм Дейкстры предполагает, что все ребра графа имеют положительный вес. Для графов, в которых есть ребра с отрицательным весом, требуется использовать более общий, но менее производительный алгоритм Беллмана — Форда (Bellman — Ford algorithm). Графы, содержащие циклы с отрицательным весом, представляют еще большую проблему. В таком графе кратчайший путь из $x$ в $y$ не определен, поскольку имеется возможность пойти из вершины $x$ по циклу с отрицательным весом и кружить по нему, делая общую стоимость сколь угодно низкой.

  Заметим, что добавление фиксированного значения к весу каждого ребра с целью сделать веса всех ребер положительными не решает проблему. В таком случае алгоритм Дейкстры будет выбирать пути с небольшим количеством ребер, даже если эти пути не были кратчайшими взвешенными путями в первоначальном графе.

- *Как поступить, если входные данные представлены не графом, а набором геометрических объектов?*

  Во многих приложениях требуется найти кратчайший путь между двумя точками при наличии геометрических объектов — например, в комнате, обставленной мебелью. Самое простое решение в таком случае заключается в преобразовании входного экземпляра задачи в граф расстояний для последующей его обработки алгоритмом Дейкстры. Вершины соответствуют граничным вершинам препятствий, а ребра определяются только между такими парами вершин, что из одной вершины «видна» другая.

  Существуют и более эффективные геометрические алгоритмы, вычисляющие кратчайший путь непосредственно по расположению препятствий. Информацию о таких геометрических алгоритмах вы найдете в *разд. 20.14* и в *подразделе «Примечания»*.

- *Является ли граф бесконтурным ориентированным графом?*

  В бесконтурных ориентированных графах кратчайший путь можно найти за линейное время. Сначала выполняем топологическую сортировку, упорядочивая вершины таким образом, чтобы все они расположились слева направо, начиная с исходной вершины $s$. Очевидно, что расстояние от вершины $s$ до самой себя — $dist(s, s)$ — равно 0. Остальные вершины обрабатываются слева направо. Обратите внимание, что

$$dist(s,j) = \min_{(i,j)\in E} \left(dist(s,i) + weight(i,j)\right),$$

  поскольку мы уже знаем кратчайший путь $dist(s, i)$ для всех вершин слева от вершины $i$. В самом деле, большинство задач динамического программирования можно сформулировать в виде задачи поиска кратчайшего пути в бесконтурном ориентированном графе. Заменив min на max, мы сможем применить тот же самый алгоритм для поиска *самого длинного пути* в бесконтурном ориентированном графе. Эту особенность можно использовать во многих приложениях — например, при составлении расписаний (см. *разд. 17.9*).

- *Требуется ли найти кратчайший путь между всеми парами точек?*

  Примитивный способ вычислить матрицу $D$ кратчайшего пути между всеми парами вершин (где $D_{ij}$ представляет расстояние между вершинами $i$ и $j$) — выполнить

алгоритм Дейкстры $n$ раз, по одному разу для каждой начальной вершины. Также можно использовать алгоритм Флойда — Уоршелла, имеющий время исполнения $O(n^3)$. Этот алгоритм действует быстрее, чем алгоритм Дейкстры, и его легче программировать. Он также работает с отрицательным весом ребер, но не с циклами. Описание алгоритма Флойда — Уоршелла вместе с реализацией представлено в *разд. 8.3.2*, а в листинге 18.4 приводится его псевдокод. Переменная $M$ обозначает матрицу весов ребер, в которой $M_{ij} = \infty$, если ребро $(i, j)$ не существует.

**Листинг 18.4. Алгоритм Флойда — Уоршелла**

```
D^0 = M
   for k = 1 to n do
      for i = 1 to n do
        for j = 1 to n do
          D_ij^k = min(D_ij^(k-1), D_ik^(k-1) + D_kj^(k-1))
Return D^n
```

Ключевым обстоятельством к пониманию алгоритма Флойда — Уоршелла является то, что $D_{ij}^k$ обозначает «длину кратчайшего пути от вершины $i$ к вершине $j$, который проходит через вершины 1, ..., $k$ как возможные промежуточные вершины». Обратите внимание, что сложность по памяти составляет всего лишь $O(n^2)$, поскольку на этапе $k$ нам требуется знать только расстояния $D^k$ и $D^{k-1}$.

- *Как найти кратчайший цикл в графе?*

  Одним из применений алгоритма поиска кратчайшего пути между всеми парами вершин является поиск кратчайшего цикла графа, называемого *обхватом* (girth).

  Алгоритм Флойда можно использовать для вычисления расстояния $d_{ii}$, где $1 \le i \le n$, что является кратчайшим путем из вершины $i$ в вершину $i$, или, иными словами, самым коротким циклом через вершину $i$.

  *Возможно*, это вам и требуется. Но кратчайший цикл через вершину $x$, вероятно, может проходить от вершины $x$ к вершине $y$ и обратно к вершине $x$, используя дважды одно и то же ребро. Цикл называется *простым*, если каждое его ребро и вершина посещаются только один раз. Подходом к поиску кратчайшего простого цикла является вычисление кратчайших путей от вершины $i$ ко всем другим вершинам с последующей проверкой наличия приемлемого ребра от каждой вершины до вершины $i$.

  Задача поиска *самого длинного цикла* графа включает задачу поиска гамильтонова цикла как частного случая (см. *разд. 19.5*), так что она является NP-полной.

Матрицу кратчайшего пути для всех пар вершин можно использовать для вычисления нескольких полезных инвариантов, связанных с центром графа $G$. *Эксцентриситетом* вершины $v$ называется кратчайший путь к самой дальней вершине от $v$. На этом понятии основаны некоторые другие инварианты. *Радиусом* графа называется минимальный из эксцентриситетов вершин графа, а вершина, на которой достигается этот минимум, — *центральной вершиной*. *Диаметром* называется наибольшее расстояние между вершинами графа.

### Реализации

Самые эффективные программы поиска кратчайшего пути написаны Эндрю Голдбергом (Andrew Goldberg) и его соавторами. Загрузить эти программы можно с веб-сайта **http://www.avglab.com/andrew/soft.html**. В частности, программа MLB на языке C++ предназначена для поиска кратчайшего пути в графах с ребрами положительного веса, выраженного целыми числами. Подробности об этом алгоритме и его реализации см. в [Gol01]. Время исполнения его обычно только в 4–5 раз больше, чем время обхода в ширину, и он может обрабатывать графы, содержащие миллионы вершин. Также существуют высокопроизводительные реализации на языке C как алгоритма Дейкстры, так и алгоритма Беллмана — Форда.

Все библиотеки графов на C++ и Java, упомянутые в *разд. 15.4*, содержат как минимум реализацию алгоритма Дейкстры. Библиотека Boost Graph Library на языке C++ (см. [SLL02]) включает обширную коллекцию алгоритмов поиска кратчайшего пути, и в том числе алгоритмы Беллмана — Форда и Джонсона. Загрузить библиотеку можно на веб-сайте **http://www.boost.org/libs/graph/doc**. Библиотека LEDA (см. *разд. 22.1.1*) содержит хорошие реализации на языке C++ всех рассмотренных здесь алгоритмов поиска кратчайшего пути, включая алгоритмы Дейкстры, Беллмана — Форда и Флойда — Уоршелла. Библиотека JGraphT (**https://jgrapht.org**) содержит реализации на языке Java как алгоритма Дейкстры, так и алгоритма Беллмана — Форда.

Соревнования DIMACS в октябре 2006 года проводились по алгоритмам поиска кратчайшего пути. При этом там обсуждались реализации эффективных алгоритмов для поиска кратчайшего пути. Соответствующие материалы можно найти на веб-сайте **http://dimacs.rutgers.edu/programs/challenge/**.

#### *Примечания*

Одним из удачных описаний алгоритмов Дейкстры (см. [Dij59]), Беллмана — Форда (см. [Bel58] и [FF62]) и Флойда — Уоршелла для поиска кратчайшего пути для всех пар вершин (см. [Flo62]) является книга [CLRS09]. Хорошие обзоры алгоритмов поиска кратчайшего пути представлены среди прочих в работах [Zwi01] и [MAR+17], а обзор геометрических алгоритмов кратчайшего пути — в книге [PN18].

Самым быстрым известным алгоритмом — с временем исполнения $O(m + n \log n)$ — для поиска кратчайшего пути из одной вершины является алгоритм Дейкстры, использующий пирамиды Фибоначчи (см. [FT87]). Экспериментальные исследования алгоритмов поиска кратчайшего пути описаны в работах [DF79] и [DGKK79]. Но эти эксперименты проводились до того, как были разработаны пирамиды Фибоначчи. Сравнительно недавнее исследование можно найти в работе [CGR99]. На практике производительность алгоритма Дейкстры можно повысить с помощью эвристических методов. В работе [HSWW05] предоставляется экспериментальное исследование взаимодействия четырех таких эвристических методов.

Приблизительный кратчайший путь между двумя точками разветвленных дорожных сетей нетрудно найти с помощью таких служб, как Google Maps. Решение этой задачи несколько отличается от решения обсуждаемых здесь задач поиска кратчайшего пути. Во-первых, затраты на предварительную обработку можно амортизировать, распределив их среди многих запросов поиска пути от одной точки до другой. Во-вторых, высокоскоростные магистрали большой протяженности могут свести задачу поиска кратчайшего пути к поиску наилучших точек для въезда на магистраль и съезда с нее. И наконец, для практических целей достаточно получить приблизительные или эвристические результаты.

Алгоритм A* выполняет поиск кратчайшего пути, выбирая первый оптимальный вариант, и сопровождает его анализом нижнего предела, чтобы установить, действительно ли найденный путь является кратчайшим путем графа.

В работе [GKW06] описана реализация алгоритма A*, способного после двух часов предварительной обработки выполнять за одну миллисекунду запросы о кратчайшем пути между двумя точками дорожной сети национального масштаба. В работе [AFGW10] выполняется анализ эвристических методов для ускорения работы алгоритмов нахождения кратчайшего пути.

Во многих приложениях, кроме оптимального пути, требуется найти альтернативные короткие пути. Отсюда возникает задача поиска $k$ кратчайших путей. Существуют варианты этой задачи, в зависимости от того, должен ли требуемый путь быть простым или он может содержать циклы. Приведенная в работе [Epp98] реализация генерирует представление этих путей за время $O(m + n \log n + k)$. Из этого представления отдельные пути можно восстановить за время $O(n)$. Новый алгоритм и экспериментальные результаты приведены в работе [HMS03].

Существуют быстрые алгоритмы для вычисления обхвата как общих, так и планарных графов (см. [IR78] и [Dji00] соответственно).

### Родственные задачи

Потоки в сетях (см. *разд. 18.9*), планирование перемещений (см. *разд. 20.14*).

## 18.5. Транзитивное замыкание и транзитивная редукция

**Вход.** Ориентированный граф $G = (V, E)$.

**Задача.** Для получения *транзитивного замыкания* создать граф $G' = (V, E')$, у которого $(i, j) \in E'$ тогда и только тогда, когда в графе $G$ существует ориентированный путь от $i$ до $j$. Для получения *транзитивной редукции* создать граф $G'$ с наименьшим количеством ребер, который содержит ориентированный путь от $i$ до $j$ тогда и только тогда, когда ориентированный путь от $i$ до $j$ существует в графе $G$ (рис. 18.5).

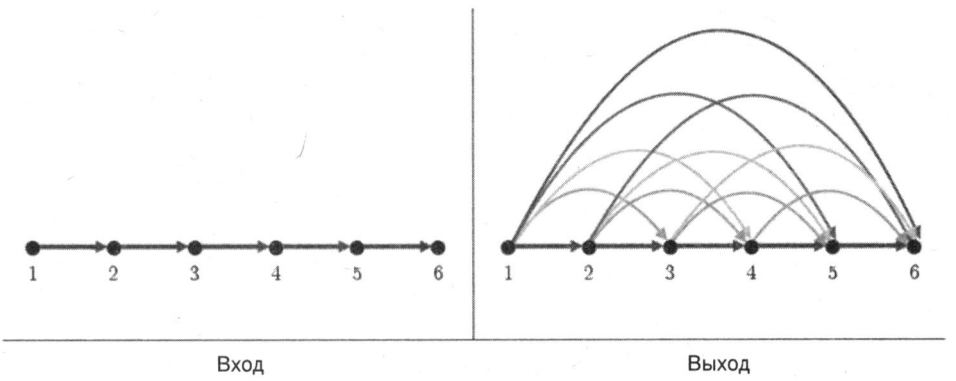



**Рис. 18.5.** Транзитивное замыкание

### Обсуждение

Транзитивное замыкание можно рассматривать как создание структуры данных, позволяющей эффективно решать запросы достижимости типа «Можно ли попасть в пункт $x$ из пункта $y$?» После создания матрицы $M$ транзитивного замыкания ответ на такие запросы можно найти за постоянное время, просто возвращая значение элемента $M[x, y]$ матрицы.

Транзитивное замыкание возникает при распространении изменений атрибутов графа $G$. Рассмотрим, например, граф, моделирующий электронную таблицу, у которого вершины представляют ячейки таблицы, а ребро $(i, j)$ соединяет соответствующие ячейки тогда и только тогда, когда результат ячейки $j$ зависит от ячейки $i$. Когда изменяется значение какой-либо ячейки, то также требуется обновить значения всех достижимых из нее ячеек. Идентификация этих ячеек осуществляется с помощью транзитивного замыкания графа $G$. Аналогичным образом многие задачи баз данных сводятся к построению транзитивных замыканий.

Для построения транзитивного замыкания применяются три основных алгоритма:

- простой алгоритм выполняет обход графа в ширину или глубину из одной из вершин и запоминает все посещенные вершины. Время работы алгоритма, выполняющего $n$ таких обходов, равно $O(n(n + m))$ и ухудшается до кубического для плотных графов. Этот алгоритм легко поддается реализации, имеет хорошую производительность на разреженных графах и, скорее всего, подходит для вашего приложения;
- алгоритм Уоршелла создает транзитивные замыкания за время $O(n^3)$, используя простой и изящный алгоритм, идентичный алгоритму Флойда для поиска кратчайшего пути между всеми парами, рассматриваемый в *разд. 18.4*. Если нас не интересует длина получившихся путей, то можно уменьшить объем требуемой памяти, используя только один бит для каждого элемента матрицы. Таким образом, $D_{ij}^k$ = true тогда и только тогда, когда вершина $j$ достижима из вершины $i$ при посещении в качестве промежуточных только вершин 1, ..., $k$;
- вычислить транзитивное замыкание можно также с помощью перемножения матриц. Пусть $M^1$ — матрица смежности графа $G$. Ненулевые значения матрицы $M^2 = M \times M$ идентифицируют все пути длиной 2 графа $G$. Обратите внимание, что $M^2[i, j] = \sum_x M[i, x] \cdot M[x, j]$, поэтому путь $(i, x, j)$ содержится в $M^2[i, j]$. Таким образом, объединение $\bigcup_i^n M^i$ выдает транзитивное замыкание $T$. Кроме того, это объединение можно вычислить всего лишь за $O(\lg n)$ матричных операций, используя быстрый алгоритм возведения в степень, рассмотренный в *разд. 16.9*.

  Для больших значений $n$ можно получить лучшую производительность, используя алгоритм Штрассена для быстрого перемножения матриц, но лично я бы не стал тратить на это свое время. Так как сложность задачи построения транзитивного замыкания доказуемо такая же, как и у задачи перемножения матриц, надежда на получение значительно более быстрого алгоритма невелика.

Для многих графов время исполнения этих трех процедур можно значительно улучшить. Вспомните, что компонентом сильной связности является набор вершин, в кото-

ром любая вершина достижима из любой другой вершины. Например, любой направленный цикл определяет сильно связный подграф. Из всех вершин в любом компоненте сильной связности должно быть достижимо одно и то же подмножество графа $G$. Таким образом, мы можем свести нашу задачу к вычислению транзитивного замыкания на графе компонентов сильной связности, который обычно имеет значительно меньшее количество ребер и вершин, чем граф $G$. Компоненты сильной связности графа $G$ можно найти за линейное время (см. *разд. 18.1*).

Транзитивная редукция (также называемая *минимальным эквивалентным ориентированным графом*) является операцией, обратной операции транзитивного замыкания, т. е. операцией уменьшения количества ребер при сохранении первоначальных свойств достижимости. Транзитивное замыкание графа $G$ идентично транзитивному замыканию транзитивной редукции этого графа. Транзитивная редукция минимизирует пространство, удаляя из графа $G$ повторяющиеся ребра, которые не влияют на достижимость. Необходимость в транзитивной редукции также возникает при рисовании графов, когда требуется убрать как можно больше подразумеваемых ребер, чтобы не загромождать рисунок.

Хотя граф $G$ может иметь только одно транзитивное замыкание, у него может быть несколько разных транзитивных редукций, в число которых входит он сам. Мы ищем наименьшую из этих редукций, но у такой задачи существуют разные формулировки:

- простой линейный алгоритм для вычисления транзитивной редукции определяет компоненты сильной связности графа $G$, заменяет каждый из них ориентированным циклом, а потом добавляет ребра, связывающие разные компоненты. Хотя это не всегда дает наименьшую возможную редукцию, на многих графах она будет довольно близка к таковой.

  Один из возможных недостатков этого эвристического алгоритма состоит в том, что он может вставить в транзитивную редукцию графа $G$ ребра, которых нет в исходном графе. Впрочем, это может и не представлять проблему в вашем конкретном приложении;

- если все ребра нашей транзитивной редукции должны существовать в графе $G$, то найти редукцию минимального размера не удастся. Попробуем разобраться, почему это так. Рассмотрим ориентированный граф, состоящий из одного компонента сильной связности, где из любой вершины можно попасть в любую другую вершину. Для такого графа наименьшей возможной транзитивной редукцией будет просто ориентированный цикл ровно из $n$ ребер. Но это окажется возможным тогда и только тогда, когда граф $G$ является гамильтоновым графом, а это доказывает, что задача поиска наименьшего такого замыкания является NP-полной.

  Эвристический подход к поиску транзитивной редукции с сохранением ребер заключается в последовательном переборе каждого ребра и его удалении, если это удаление не меняет исходное транзитивное замыкание. Эффективная реализация этого процесса позволит минимизировать время, затрачиваемое на проверку достижимости. Обратите внимание, что ориентированное ребро $(i, j)$ можно удалить в любом случае, когда от вершины $i$ к вершине $j$ имеется другой путь, минующий это ребро;

- редукцию минимального размера, используя ребра между произвольными парами вершин, можно найти за время $O(n^3)$. Но для большинства приложений, скорее всего, будет достаточно описанного ранее простого, но действенного алгоритма, благодаря его более высокой эффективности и более легкой программируемости.

### Реализации

Алгоритмы построения транзитивного замыкания и выполнения редукции, содержащиеся в библиотеке Boost Graph Library, отличаются особенно качественной технической реализацией (см. [SLL02]). Дополнительная информация и исходные коды доступны на веб-сайте библиотеки **http://www.boost.org/libs/graph**. Библиотека LEDA (см. *разд. 22.1.1*) содержит реализации на языке C++ алгоритмов для вычисления как транзитивного замыкания, так и транзитивной редукции. Подробности о библиотеке LEDA см. в [MN99]. Обширная библиотека алгоритмов для работы с графами JGraphT (**https://jgrapht.org/**) содержит реализации обеих алгоритмов на языке Java.

Библиотека Combinatorica (см. [PS03]) содержит реализации (на языке пакета Mathematica) алгоритмов поиска транзитивного замыкания и транзитивной редукции. Подробности приведены в *разд. 22.1.8*.

#### *Примечания*

Транзитивное замыкание и транзитивная редукция обсуждаются в работе [vL90a]. Эквивалентность перемножения матриц и вычисления транзитивного замыкания была доказана Фишером (Fischer) и Мейером (Meyer) в работе [FM71], а описания алгоритмов можно найти в книге [AHU74].

В последнее время наблюдался повышенный интерес исследователей к транзитивному замыканию, большая часть которого отражена в работе [Nuu95]. Пеннер (Penner) и Прасанна (Prasanna) (см. [PP06]) улучшили производительность алгоритма Уоршелла (см. [War62]) приблизительно вдвое посредством реализации, чувствительной к кэшированию.

Доказательство эквивалентности транзитивного замыкания и транзитивной редукции было представлено в работе [AGU72]. Результаты экспериментальных исследований алгоритмов вычисления транзитивного замыкания можно найти в [Nuu95], [PP06] и [SD75]. Разреженные транзитивные сокращения часто содержат длинные пути. В задачах *каркаса транзитивного замыкания* (transitive closure spanner) требуется найти небольшие уменьшения диаметра, не превышающие $k$ (см. [BGJ+12]).

Важным аспектом оптимизации запросов к базам данным является оценка размера транзитивного замыкания. Алгоритм решения этой задачи за линейное время представлен в работе [Coh94].

### Родственные задачи

Компоненты связности (см. *разд. 18.1*), поиск кратчайшего пути (см. *разд. 18.4*).

## 18.6. Паросочетание

**Вход.** Граф (возможно, взвешенный) $G = (V, E)$.

**Задача.** Найти наибольшее подмножество ребер $E'$ множества $E$, для которого каждая вершина множества $V$ инцидентна самое большее одному ребру из подмножества $E'$ (рис. 18.6).

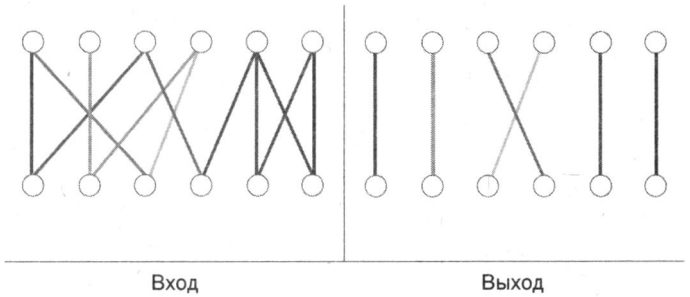



**Рис. 18.6.** Поиск паросочетаний

### Обсуждение

Допустим, что мы руководим группой рабочих, и каждый из них умеет выполнять определенное подмножество операций, необходимых для выполнения работы. Какое задание нужно сопоставить с каждым из них? Для решения этой задачи мы создадим граф, в котором вершины представляют всех рабочих и все операции, и соединим ребрами рабочих с операциями, которые они могут выполнять. Каждому рабочему назначается одна операция. Нашей целью является наибольший набор ребер, при котором не повторяются ни рабочие, ни операции, — т. е. паросочетание.

*Паросочетание* — это мощный алгоритмический инструмент, и даже удивительно, что оптимальные паросочетания могут быть найдены достаточно эффективным образом за полиномиальное время. Необходимость в применении паросочетаний на практике возникает часто.

Переженить группу мужчин и женщин таким образом, чтобы каждая пара была счастливой, — еще один пример практического применения задачи паросочетания. В соответствующем графе каждая подходящая пара соединяется ребром. В синтетической биологии (см. [MPC+06]) требуется перемешать символы в строке $S$ таким образом, чтобы максимизировать количество перемещенных символов. Например, символы строки *aaabc* можно перемешать так, что получится строка *bcaaa*, в которой только средний символ *a* занимает свою исходную позицию. Это вариант задачи паросочетания, в котором вместо мужчин фигурируют символы строки, а вместо женщин — позиции в строке (от 1 до $|S|$). Ребра соединяют символы с позициями в строке, в которых первоначально находились другие символы.

Эта основная структура паросочетаний может быть улучшена несколькими способами, и при этом останется все той же задачей о *назначениях*. Вы должны ответить на следующие вопросы.

- *Является ли граф двудольным?*

  Во многих задач о назначениях используются двудольные графы — как, например, в классической задаче о сопоставлении жениха с невестой. Но современные браки представляются паросочетаниями на более общих, недвудольных графах. Для решения задачи паросочетания на общих графах также существуют эффективные алгоритмы, но методы для двудольных графов проще и быстрее.

- *Можно ли некоторым рабочим назначать не одно, а несколько заданий?*

  Естественные обобщения задачи о назначениях включают поручение некоторым конкретным рабочим нескольких заданий или (что эквивалентно) назначение нескольких рабочих на одно задание. Такие требования можно моделировать созданием стольких копий рабочего, сколько заданий мы хотим ему поручить. Как раз этот подход (с использованием нескольких экземпляров одинаковых букв) применялся в приведенном ранее примере с перестановками символов строки.

- *Является ли граф взвешенным?*

  Многие рассмотренные до сих пор приложения паросочетаний основаны на невзвешенных графах. В них мы искали паросочетание *максимальной мощности* — в идеале *совершенное* паросочетание, в котором каждая вершина паросочетания сопоставляется с другой.

  Но для других приложений нам нужно дополнить каждое ребро весом — например, отражающим способность рабочего к выполнению того или иного задания или учитывающим, насколько человеку $x$ нравится человек $y$. Тогда задача превращается в задачу создания паросочетания с максимальным *весом* — т. е. нахождения набора независимых ребер с максимальной общей стоимостью.

Эффективные алгоритмы для построения паросочетаний работают, создавая в графах *увеличивающие пути*. Для частичного паросочетания $M$ в графе $G$ увеличивающим путем является путь $P$ из ребер, которые то входят в паросочетание $M$, то выходят из него. Имея такой увеличивающий путь, мы всегда можем расширить паросочетание на одно ребро, заменив четные ребра пути $P$ из $M$ нечетными ребрами этого пути. Согласно теореме Берга, паросочетание является максимальным тогда и только тогда, когда оно не содержит ни одного увеличивающего пути. Поэтому мы можем создавать паросочетания максимальной мощности, выполняя поиск увеличивающих путей и прекращая поиск, когда больше нет таких путей.

Задача поиска паросочетания в общих графах более сложная, чем в двудольных графах, поскольку в них возможны увеличивающие пути, содержащие циклы нечетной длины (т. е. с одной и той же начальной и конечной вершиной). Такие циклы невозможны в двудольных графах, которые не содержат их по определению.

Стандартные алгоритмы вычисления паросочетания в двудольном графе основаны на потоках в сети. В них применяется простое преобразование двудольного графа в эквивалентный потоковый граф. Реализация такого подхода приведена в *разд. 8.5*.

Однако необходимо иметь в виду, что для решения задач паросочетания во взвешенных графах требуются другие подходы, наиболее известным из которых является венгерский алгоритм.

### Реализации

Эндрю Голдберг (Andrew Goldberg) в соавторстве с другими исследователями разработал высокопроизводительные коды для вычисления паросочетаний как во взвешенных, так и в невзвешенных двудольных графах. В частности, Голдберг и Кеннеди написали на языке C программу CSA для вычисления паросочетаний во взвешенных двудольных графах, основанную на потоках в сети, масштабирующих стоимость (см. [GK95]). Бо-

лее быстрая программа BIM для вычисления паросочетаний в невзвешенных двудольных графах, основанная на методах с использованием увеличивающих путей, представлена в работе [CGM+98]. Обе программы можно загрузить для некоммерческого использования с веб-сайта **http://www.avglab.com/andrew/soft.html**.

Первое соревнование по реализации алгоритмов DIMACS (см. [JM93]) было посвящено в основном потокам в сетях и паросочетаниям. На этом соревновании было выявлено несколько удачных генераторов экземпляров задач, а также ряд реализаций алгоритмов для вычисления паросочетаний максимальной мощности и паросочетаний с максимальным весом. Эти программы можно загрузить по адресу **http://dimacs.rutgers.edu/archive/Challenges/**. В их числе вы найдете:

- решатель на языке C, разработанный Эдвардом Ротбергом (Edward Rothberg), для вычисления паросочетаний максимальной мощности, реализующий алгоритм Габова (Gabow's algorithm) с временем исполнения $O(n^3)$;
- решатель, разработанный также Эдвардом Ротбергом, для вычисления паросочетаний с максимальным весом. Это более медленный решатель, но более общий, чем только что упомянутый решатель этого же автора.

Библиотека LEDA (см. *разд. 22.1.1*) содержит эффективные реализации на языке C++ для вычисления паросочетаний как максимальной мощности, так и максимального веса, как на двудольных, так и на общих графах. Эффективную программу Blossom IV (см. [CR99]) на языке C для вычисления совершенных паросочетаний минимального веса можно загрузить с веб-сайта **http://www.math.uwaterloo.ca/~bico//software.html**. Реализацию для вычисления паросочетаний максимальной мощности (см. [KP98]) в общих графах за время $O(mn\alpha(m, n))$ можно загрузить с веб-сайта **http://www.cs.arizona.edu/~kece/Research/software.html**.

База графов Stanford GraphBase (см. *разд. 22.1.7*) содержит реализацию венгерского алгоритма для вычисления паросочетаний в двудольных графах. В целях визуализации взвешенных двудольных графов Дональд Кнут использует оцифрованную версию картины «Мона Лиза» и осуществляет в ней поиск пикселов максимальной яркости в отдельных строчках и столбцах. Паросочетание также используется для построения оригинальных портретов в стиле «домино».

### Примечания

Книга Ловаша (Lovasz) и Пламмера (Plummer) [LP09] представляет собой полный справочник по теории паросочетаний и алгоритмам их вычисления. Среди обзорных статей по алгоритмам вычисления паросочетаний особого внимания заслуживает работа [Gal86]. Хорошие описания алгоритмов, использующих потоки в сети для вычисления паросочетаний в двудольных графах, содержатся в работах [CLRS01], [Eve11] и [Man89], а венгерского алгоритма — в работах [Law11] и [PS98]. Самый лучший алгоритм для вычисления максимальных паросочетаний в двудольных графах, разработанный Хопкрофтом (Hopcroft) и Карпом (Karp) (см. [HK73]), последовательно находит кратчайшие увеличивающие пути (вместо использования сетевого потока) и исполняется за время $O(\sqrt{n}m)$. Время исполнения венгерского алгоритма равно $O(n(m + n \log n))$.

Алгоритм Эдмондса (Edmonds) (см. [Edm65]) для вычисления паросочетаний максимальной мощности представляет большой исторический интерес, поскольку он вызывает вопросы о том, какие задачи можно решить за полиномиальное время. Описания алгоритма

Эдмондса можно найти в [Law11], [PS98] и [Tar83]. Время исполнения самого лучшего известного алгоритма вычисления общих паросочетаний равно $O(\sqrt{n}m)$ (см. [MV80]).

Вернемся к задаче паросочетания мужчин и женщин. Предположим, входной экземпляр содержит ребра $(B_1, G_1)$ и $(B_2, G_2)$, причем в действительности мужчина $B_1$ и женщина $G_2$ предпочитают друг друга своим текущим партнерам. В реальной жизни эти двое, скорее всего, сойдутся, расторгнув свои браки. Паросочетание, не имеющее таких пар, называется *устойчивым* (stable). Всестороннее изучение устойчивых паросочетаний содержится в книге [GI89]. Интересно, что независимо от многообразия взаимных предпочтений мужчин и женщин, всегда существует хотя бы одно устойчивое паросочетание. Более того, такое паросочетание можно найти за время $O(n^2)$ (см. статью [GS62]).

В Интернете задачи сопоставления появляются, когда выбор ребер нужно сделать, не имея полной информации о графе. Такие задачи возникают, в частности, в рекламе в Интернете. Компетентное исследование таких задач представлено в [M+13].

В двудольных графах размер максимального паросочетания равен размеру минимального вершинного покрытия. Это означает, что на двудольных графах задачу о минимальном вершинном покрытии и задачу о максимальном независимом множестве можно решить за полиномиальное время.

### Родственные задачи

Эйлеров цикл (см. *разд. 18.7*), потоки в сети (см. *разд. 18. 9*).

## 18.7. Задача поиска эйлерова цикла и задача китайского почтальона

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти самый короткий маршрут, который проходит по каждому ребру графа $G$ (рис. 18.7).

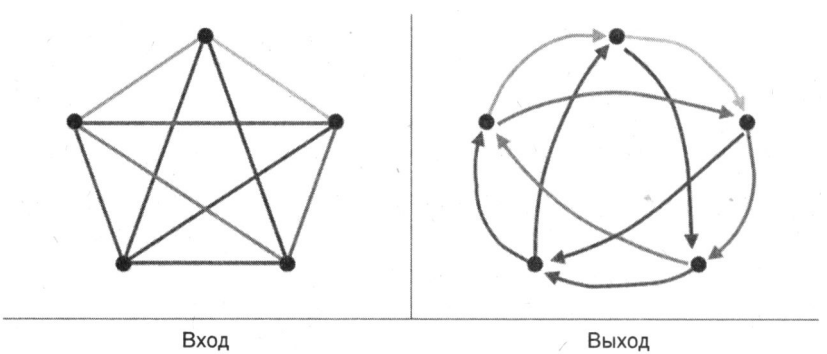



**Рис. 18.7.** Эйлеров цикл

### Обсуждение

Допустим, что вам дали задание разработать ежедневный маршрут для мусоровозов, или для снегоуборочных машин, или для почтальонов. В каждом случае необходимо целиком пройти каждую улицу по крайней мере один раз. Для эффективного выполне-

ния задания нам нужно минимизировать время прохождения маршрута или (что равносильно) общее расстояние или количество ребер графа, по которым осуществляется обход.

Рассмотрим другой пример: проверку системы меню телефонного автоматизированного справочника. Каждая опция «Нажмите такую-то кнопку для получения такой-то информации» рассматривается как ориентированное ребро между двумя вершинами графа. Тестировщик меню ищет самый эффективный способ обхода этого графа, стараясь пройти по каждому ребру, по крайней мере, один раз.

Обе эти задачи являются вариантами задачи поиска эйлерова цикла, которую лучше всего сравнить с головоломкой, где нужно нарисовать геометрическую фигуру, не отрывая карандаш от бумаги и не проводя его по уже нарисованным линиям. То есть найти путь или цикл в графе, который проходит по каждому ребру только один раз.

Приведем признаки наличия в графе эйлерова цикла:

- *неориентированный* граф содержит эйлеров цикл тогда и только тогда, когда граф связный и все его вершины имеют четную степень;
- *ориентированный* граф содержит эйлеров цикл тогда и только тогда, когда граф сильно связный и все вершины имеют одинаковые степень захода и степень исхода.

Для эйлеровых *путей*, покрывающих все ребра, но не возвращающихся в начальную вершину, условия степени вершины ослабляются. *Неориентированный* связный граф содержит эйлеров путь тогда и только тогда, когда все вершины, кроме двух, имеют четную степень. Эти две вершины и будут начальной и конечной точками любого пути. Наконец, *ориентированный* связный граф содержит эйлеров путь от вершины $x$ к вершине $y$ тогда и только тогда, когда все его вершины имеют одинаковые степень захода и степень исхода — с тем исключением, что вершина $x$ имеет степень захода на единицу меньше, чем степень исхода, а вершина $y$ имеет степень захода на единицу больше, чем степень исхода.

Эти свойства эйлеровых графов позволяют с легкостью проверить наличие пути или цикла: сначала проверяем граф на связность, выполнив обход в глубину или ширину, а потом подсчитываем количество вершин с нечетной степенью. Поиск цикла также можно выполнить явно за линейное время, используя алгоритм Хирхольцера (Hierholzer), следующим образом. Посредством обхода в глубину находим в графе произвольный цикл. Удаляем этот цикл из графа и повторяем процедуру до тех пор, пока все множество ребер не будет разбито на циклы, не имеющие общих ребер. Так как удаление цикла уменьшает степень каждой вершины на четное число, то оставшийся граф продолжает удовлетворять все тем же эйлеровым степенным условиям. Эти циклы должны иметь общие вершины (поскольку граф является связным), и поэтому их можно объединить в виде «восьмерок» по любой общей вершине. Объединяя все извлеченные циклы, мы создаем один цикл, содержащий все ребра.

Эйлеров цикл, если такой существует в графе, решает задачу построения маршрута снегоуборочной машины, т. к. длина любого пути, проходящего по каждому ребру только один раз, должна быть минимальной. Но на практике дорожные сети редко удовлетворяют эйлеровым степенным условиям. Тогда нам приходится решать более общую задачу — *задачу китайского почтальона*, в которой требуется найти мини-

мальный цикл, проходящий через каждое ребро по крайней мере один раз. Этот минимальный цикл никогда не пройдет более двух раз ни по какому ребру, так что удовлетворительный маршрут удается найти для любой дорожной сети.

Оптимальный маршрут китайского почтальона можно создать, добавив соответствующие ребра в граф $G$, чтобы сделать его эйлеровым. Добавив путь между двумя вершинами с нечетной степенью, мы превращаем их в вершины с четной степенью, что приближает граф $G$ к тому, чтобы он стал эйлеровым.

Задача поиска наилучшего множества кратчайших путей для добавления к графу $G$ сводится к поиску совершенного паросочетания с минимальным весом в специальном графе $G'$. Для неориентированных графов вершины графа $G'$ соответствуют вершинам нечетной степени графа $G$, где вес ребра $(i, j)$ определяется как длина кратчайшего пути от вершины $i$ к вершине $j$ в графе $G$. Для ориентированных графов вершины графа $G'$ соответствуют вершинам графа $G$, несбалансированным по степени, при этом все ребра в графе $G'$ направлены из вершин с нехваткой степени исхода в вершины с нехваткой степени захода. Таким образом, для ориентированного графа $G$ будет достаточно использовать алгоритмы для вычисления паросочетания в двудольных графах. Когда граф становится эйлеровым, то оптимальный цикл можно получить за линейное время.

### Реализации

Реализацию алгоритма поиска эйлерова цикла содержат многие библиотеки графов, но реализации решений задачи китайского почтальона встречаются менее часто. Я рекомендую реализацию на языке Java для решения задачи китайского почтальона, разработанную Тимблби (Thimbleby), — см. [Thi03]. Программу можно загрузить с вебсайта **http://www.harold.thimbleby.net/cpp/index.html**. Библиотека JGraphT (**https://jgrapht.org**) содержит реализацию алгоритма Хирхольцера на языке Java.

В библиотеке GOBLIN (**http://goblin2.sourceforge.net**) вы найдете обширную коллекцию процедур на языке C++ для всех стандартных задач оптимизации на графах, включая процедуру решения задачи китайского почтальона как на ориентированных, так и на неориентированных графах. Библиотека LEDA (см. *раздел 22.1.1*) предоставляет необходимые инструменты эффективной реализации решателей для задач поиска эйлерова цикла, паросочетаний и кратчайших путей в двудольных и общих графах.

Библиотека Combinatorica (см. [PS03]) содержит реализации (на языке пакета Mathematica) для решения задач поиска эйлеровых циклов и последовательностей де Брейна (de Bruijn sequences). Подробности см. в *разд. 22.1.8*.

#### Примечания

История теории графов начинается в 1736 году, когда Леонард Эйлер взялся за решение задачи о семи мостах Кенигсберга. Город Кенигсберг (в настоящее время Калининград) расположен на берегах реки. Во времена Эйлера оба берега и два острова реки соединялись семью мостами. Эту планировку можно смоделировать в виде мультиграфа с четырьмя вершинами и семью ребрами. Эйлер захотел узнать, возможно ли пройти по всем мостам только по одному разу и возвратиться в исходную точку (и впоследствии такой маршрут получил название эйлерова цикла). Он доказал, что искомый маршрут невозможен, поскольку все четыре вершины имели нечетную степень. Мосты, по которым ходил Эйлер, были разрушены во время Второй мировой войны. История возникновения задачи рассказана в книге [BLW76].

Из недавних работ по задачам маршрутизации транспортных средств и граничной маршрутизации можно назвать книги [CL13] и [TV14]. Описания алгоритмов с линейным временем исполнения для создания эйлеровых циклов (см. [Ebe88]) вы найдете в работах [Eve11] и [Man89]. Простым и элегантным методом создания эйлеровых циклов является алгоритм Флери (Fleury) — см. [Luc91]: начинаем обход с любой вершины и удаляем пройденные ребра. Единственным критерием выбора следующего ребра является отказ от прохода по мосту (т. е. ребру, удаление которого разъединит граф) до тех пор, пока не останется других вариантов.

Метод поиска эйлерова пути играет важную роль в параллельных алгоритмах на графах. Многие параллельные алгоритмы начинаются с построения остовного дерева, для которого потом устанавливается корень по методу поиска эйлерова пути. Описание параллельных алгоритмов приведено в таких учебниках, как [J92], а результаты практического применения — в работе [CB04]. Для подсчета эйлеровых циклов в графах существуют эффективные алгоритмы (см. [HP73]).

Задача поиска кратчайшего маршрута, проходящего через все ребра графа, впервые была представлена китайским ученым Кваном (Kwan) (см. [Kwa62]), что и определило ее название — задача китайского почтальона. Решение задачи китайского почтальона посредством алгоритма поиска паросочетаний в двудольных графах разработано Эдмондсом (Edmonds) и Джонсоном (Johnson) — см. [EJ73]. Этот алгоритм работает как с ориентированными, так и с неориентированными графами, но для смешанных графов задача является NP-полной (см. [Pap76a]). Смешанные графы содержат как ориентированные, так и неориентированные ребра. Описание алгоритма решения задачи китайского почтальона приводится в [Law11].

Последовательность де Брейна (de Bruijn) $S$ протяженностью $k$ на алфавите $\Sigma$ из $\alpha$ символов представляет собой циклическую строку длиной $\alpha^k$, содержащую все строки длиной $k$ как подстроки последовательности $S$ — каждую только один раз. Например, для $k = 3$ и $\Sigma = \{0, 1\}$ циклическая строка 00011101 содержит по порядку подстроки 000, 001, 011, 111, 110, 101, 010, 100. Последовательности де Брейна можно рассматривать как «руководство для вскрытия сейфа», которое предоставляет самый короткий набор поворотов ручки кодового замка, причем $\alpha$ дает количество позиций, достаточное для перебора всех комбинаций длиной $n$.

Эти последовательности можно генерировать, создав ориентированный граф, все вершины которого представляют строки $\alpha^{n-1}$ длиной $n - 1$, и в котором ребро $(u, v)$ существует тогда и только тогда, когда $u = s_1s_2...s_{n-1}$ и $v = s_2...s_{n-1}s_n$. Любой эйлеров цикл на таком графе описывает последовательность де Брейна. Описания последовательностей де Брейна и способы их построения приведены в [Eve11] и [PS03].

Интересный алгоритм на основе эйлерова цикла для оптимизации узоров вышивания рассматривается в [AHK+08].

### Родственные задачи

Паросочетание (см. *разд. 18.6*), гамильтонов цикл (см. *разд. 19.5*).

## 18.8. Реберная и вершинная связность

**Вход.** Граф $G$ и, возможно, пара вершин $s$ и $t$.

**Задача.** Найти наименьшее подмножество вершин (или ребер), удаление которых разъединит граф $G$ или, как вариант, отделит вершину $s$ от вершины $t$ (рис. 18.8).

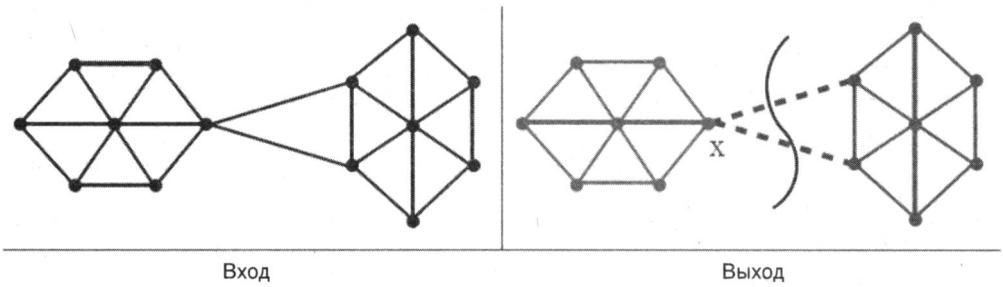



**Рис. 18.8.** Выяснение числа реберной связности

### Обсуждение

Вопрос о связности графов часто возникает в задачах, имеющих отношение к надежности сетей. Например, в случае сетей связи число вершинной связности — это наименьшее количество коммутационных станций, которые нужно вывести из строя, чтобы разорвать сеть, т. е. сделать невозможной связь между любой парой исправных коммутационных станций. А число реберной связности в этом случае — наименьшее количество линий связи, которые нужно разорвать, чтобы достичь этой же цели.

Итак, *числом реберной (вершинной) связности* графа $G$ называется наименьшее количество ребер (вершин), удаление которых нарушает связность графа. Между этими двумя величинами — количеством ребер и вершин — существует тесная связь. Число вершинной связности всегда меньше или равно числу реберной связности, поскольку удаление одной вершины из каждого ребра разреза нарушает связность графа. Но возможно существование и меньшего подмножества вершин. Верхним пределом, как для реберной, так и для вершинной связности, является минимальная степень вершины, поскольку удаление всех смежных с ней вершин (или ребер, связывающих ее с соседними вершинами) отсоединяет компонент, состоящий из единственной вершины, от остальной части графа.

Представляет интерес поиск ответов на следующие вопросы.

- *Является ли граф связным?*

  Проверка связности графа — весьма простая задача. Все компоненты связности можно найти за линейное время посредством обхода в глубину или ширину (см. *разд. 18.1*). Для ориентированных графов следует выяснить, является ли граф *сильно связным* — т. е. существует ли в нем ориентированный путь между каждой парой вершин. В *слабо связном* графе возможно существование путей к узлам, из которых нет возврата.

- *Содержит ли граф «слабое звено»?*

  Граф $G$ называется *двусвязным*, если он не содержит вершины, удаление которой нарушило бы его связность. Вершина, удаление которой приводит к потере связности графа, называется *шарниром*. *Мост* представляет собой аналогичное понятие для ребер.

  Самый простой алгоритм идентификации шарниров (или мостов) — удалять по одной вершине (или ребру) и после каждого удаления проверять оставшийся граф на

связность с помощью обхода в глубину или ширину. Для обеих задач существуют более сложные алгоритмы на основе обхода в глубину. Реализацию их см. в *разд. 7.9.2*.

- *Как разделить граф на равные части?*

  Нередко требуется найти небольшой разрез, который разделяет граф на приблизительно равные части. Например, мы хотим разделить компьютерную программу на две части, которые проще сопровождать. Для этого мы можем создать граф, вершины которого будут соответствовать процедурам программы. Соединим ребрами взаимодействующие вершины (процедуры, одна из которых вызывает другую). Далее нам нужно разделить все процедуры на наборы приблизительно одинакового размера так, чтобы минимизировать количество пар взаимодействующих процедур, расположенных по разные стороны от линии раздела.

  Эта задача называется задачей *разбиения графа* — она рассматривается в *разд. 19.6*. Хотя задача разбиения графа является NP-полной, для ее решения существуют приемлемые эвристические методы.

- *Можно ли разбивать граф произвольным образом, или требуется разбить конкретную пару вершин?*

  Существуют две разновидности общей задачи связности. В одной требуется найти наименьший разрез для всего графа, а в другой — наименьший разрез, чтобы отделить вершину $s$ от вершины $t$. Любой алгоритм поиска связности между вершинами $s$ и $t$ можно использовать $(n-1)$ раз в качестве алгоритма для проверки общей связности, т. к. после удаления любого разреза вершина $v_1$ и хотя бы одна из остальных $n-1$ вершин должны оказаться в разных компонентах.

Компоненты связности можно найти с помощью методов, применяемых для решения задач о потоках в сети, в которых взвешенный граф рассматривается как сеть труб, где каждое ребро/труба имеет максимальную пропускную способность. Целью задачи является максимизировать поток между двумя конкретными вершинами графа. Максимальный поток между вершинами $v_i$ и $v_j$ в графе $G$ в точности равен весу наименьшего набора ребер, которые нужно удалить для разъединения этих вершин. Таким образом, число реберной связности можно найти, минимизировав поток между вершиной $v_1$ и каждой вершиной из остальных $n-1$ вершин невзвешенного графа $G$.

Вершинная связность описывается *теоремой Менгера* (Menger's theorem), которая утверждает, что граф является $k$-связным тогда и только тогда, когда каждую пару вершин связывают по меньшей мере $k$ путей, не имеющих общих вершин. Здесь тоже можно использовать задачу о потоках в сети, поскольку поток объемом $k$ между парой вершин свидетельствует о наличии $k$ путей, не имеющих общих ребер.

Чтобы применить теорему Менгера, создаем такой граф $G'$, в котором любой набор путей, не имеющих общих ребер, соответствует путям, не имеющим общих вершин, в графе $G$. Для этого каждую вершину $v_i$ графа $G$ мы заменяем двумя вершинами $v_{i,1}$ и $v_{i,2}$, соединяя их ребром $(v_{i,1}, v_{i,2})$ в графе $G'$. Кроме этого, каждое ребро $(v, x) \in G$ заменяем ребрами $(x_0, y_1)$ и $(x_1, y_0)$ в графе $G'$. В результате каждому из путей, не имеющих общих вершин в графе $G$, соответствуют два пути, не имеющих общих ребер в графе $G'$. По существу, максимальный поток графа $G'$ обладает вдвое большей вершинной связностью, чем граф $G$.

### Реализации

Коллекция MINCUTLIB содержит коды нескольких высокопроизводительных алгоритмов поиска разреза, включая алгоритмы, использующие метод потоков в сети и метод сжатия. Эти реализации были разработаны Чекури (Chekuri) и его коллегами в ходе замечательного экспериментального исследования указанных алгоритмов и эвристических методов для ускорения их выполнения (см. [CGK+97]). Для некоммерческого использования программы можно загрузить с веб-сайта **http://www.avglab.com/andrew/soft.html**.

Большинство библиотек структур данных для представления графов, рассматриваемых в *разд. 18.1*, содержат процедуры проверки на связность и двусвязность. Библиотека процедур на языке C++ Boost Graph Library (см. [SLL02]) содержит процедуры проверки реберной связности. Загрузить библиотеку можно с веб-сайта **http://www.boost.org/libs/graph/doc**.

Библиотека GOBLIN (**http://goblin2.sourceforge.net/**) содержит обширную коллекцию процедур для всех стандартных задач оптимизации на графах, включая процедуры выяснения реберной и вершинной связности. Библиотека LEDA на языке C++ (см. *разд. 19.1.1*) предоставляет широкую поддержку для выяснения низкоуровневой связности (как двусвязных, так и трехсвязных компонент), реберной связности и минимального разреза.

#### Примечания

Хорошие описания использования потоков в сети для выяснения реберной и вершинной связности можно найти в [Eve11] и [PS03], а также в книге [NI08]. Правильность этих алгоритмов основана на теореме Менгера (см. [Men27]), утверждающей, что связность определяется количеством путей, не имеющих общих ребер, и путей, не имеющих общих вершин и связывающих пару вершин. Теорема о минимальном потоке и максимальном разрезе была доказана Фордом (Ford) и Фалкерсоном (Fulkerson) — см. [FF62].

Самые быстрые алгоритмы поиска минимального разреза и реберной связности основаны на методе сжатия графа, а не на методе потоков в сети. Сжатие ребра $(x, y)$ в графе $G$ соединяет две инцидентные ему вершины в одну, удаляя петли, но оставляя мультиребра. Любая последовательность таких сжатий может увеличить (но не уменьшить) минимальный разрез в графе $G$, и не меняет разрез, если не затрагивается ребро разреза. Каргер (Karger) предоставил изящный рандомизированный алгоритм поиска минимального разреза, основанный на наблюдении, что вероятность изменения минимального разреза в течение любой случайной последовательности удалений является ничтожно малой. Отличный обзор рандомизированных алгоритмов, включая алгоритм Каргера, представлен в книге [MR95].

Самая быстрая версия алгоритма Каргера имеет время исполнения $(m \lg^3 n)$ (см. [Kar00]). Также известны немного более быстрые детерминистские алгоритмы (см. [HRW17]). Результаты экспериментальных сравнений алгоритмов поиска минимальных разрезов представлены в работах [CGK+97] и [HNSS18].

Методы поиска минимального разреза нашли применение в области распознавания образов — в частности, для сегментации изображений. В своей работе [BK04] Бойков и Колмогоров приводят экспериментальную оценку алгоритмов поиска минимального разреза в контексте такого применения.

Матулой (Matula) был разработан алгоритм, не использующий методы потоков в сети, для проверки графа на реберную $k$-связность за время $O(kn^2)$. Для определенных небольших

значений $k$ существуют более быстрые алгоритмы выяснения $k$-связности. Все трехсвязные компоненты графа можно сгенерировать за линейное время (см. [HT73a]), в то время как 4-связность можно проверить за время $O(n^2)$.

### Родственные задачи

Компоненты связности (см. *разд. 18.1*), потоки в сети (см. *разд. 18.9*), разбиение графов (см. *разд. 19.6*).

## 18.9. Потоки в сети

**Вход.** Ориентированный граф $G$, каждое ребро которого $e = (i, j)$ имеет пропускную способность $c_e$, и две вершины: исток $s$ и сток $t$.

**Задача.** Найти максимальный поток, который можно направить из вершины $s$ в вершину $t$, соблюдая ограничивающие условия пропускной способности каждого ребра (рис. 18.9).

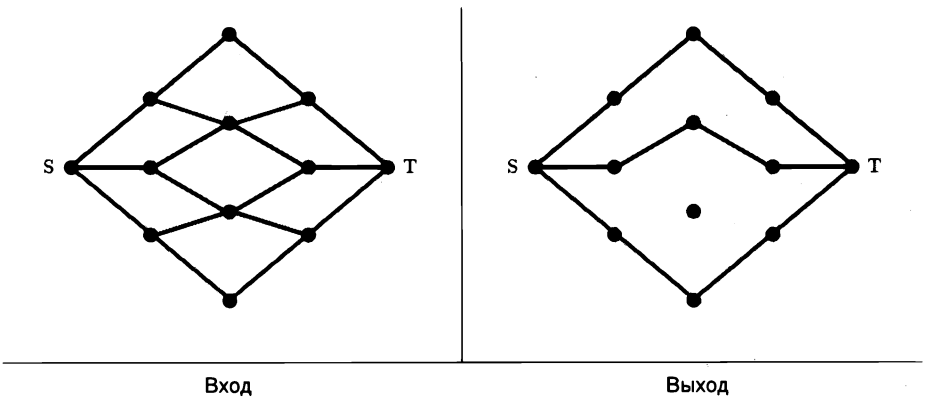



**Рис. 18.9.** Потоки в сети

### Обсуждение

Область применения задачи о потоках в сети гораздо шире, чем прокладка водопровода. Задача о потоках возникает при поиске наиболее экономически эффективного способа транспортировки продукции между множеством предприятий и множеством магазинов или при распределении ресурсов в сетях связи.

Но реальная мощь задачи о потоках в сети проявляется в том, что большое количество возникающих на практике задач линейного программирования можно смоделировать в виде задач о потоках в сети, включая несколько задач на графах, рассмотренных в этой книге: паросочетание в двудольных графах, нахождение кратчайшего пути, реберная и вершинная связность. Соответствующие алгоритмы позволяют решить эти задачи намного быстрее, чем применение методов линейного программирования общего назначения.

Вы должны развить в себе умение распознавать возможность моделирования стоящей перед вами задачи в виде задачи о потоках в сети. Такое умение требует практического

опыта и теоретических знаний. Я рекомендую сначала создать для вашей задачи модель из области линейного программирования, а потом сравнить ее с моделями для двух основных классов задач о потоках в сети.

- *Максимальный поток.*

  Здесь мы хотим найти максимальный объем потока из вершины $s$ к вершине $t$, соблюдая ограничивающие условия пропускной способности ребер графа $G$. Пусть переменная $x_{ij}$ обозначает объем потока, проходящего из вершины $i$ через ориентированное ребро $(i, j)$. Так как объем потока, проходящего через это ребро, ограничен его пропускной способностью $c_{ij}$, то

  $$0 \le x_{ij} \le c_{ij} \text{ для } 1 \le i, j \le n.$$

  Кроме того, про каждую вершину, не являющуюся ни истоком, ни стоком, можно сказать, что в нее входит поток такого же объема, какой выходит из нее, поэтому

  $$\sum_{j=1}^{n} x_{ji} - \sum_{j=1}^{n} x_{ij} = 0 \text{ для всех } 1 \le i \le n.$$

  Нам нужно найти такой набор значений, который максимизирует поток, идущий в вершину $t$, а именно $\sum_{i=1}^{n} x_{it}$.

- *Поток минимальной стоимости.*

  Здесь каждое ребро $(i, j)$ имеет дополнительный параметр, а именно стоимость $d_{ij}$ перемещения единицы потока от вершины $i$ к вершине $j$. Установлен и целевой объем потока $f$, который мы хотим направить от вершины $s$ к вершине $t$ с минимальной общей стоимостью. Следовательно, нам требуется найти такой набор значений, который минимизирует следующую формулу:

  $$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij} \cdot x_{ij},$$

  соблюдая ограничивающие условия реберной и вершинной пропускной способности для максимального потока, а также дополнительное ограничивающее условие, что $\sum_{i=1}^{n} x_{it} = f$.

При этом учитываются следующие особенности:

- *наличие нескольких истоков и/или стоков.*

  Это обстоятельство не является проблемой, поскольку мы можем модифицировать сеть, добавив в нее вершину-суперисток, которая питает все истоки, и вершину-суперсток, которая поглощает все стоки;

- *пропускная способность каждого ориентированного ребра равна либо 0, либо 1.*

  Для таких сетей существуют более быстрые алгоритмы. Подробности вы найдете в *подразделе «Примечания»*;

- *одинаковая стоимость всех ребер.*

  В таком случае используйте более простые и быстрые алгоритмы для решения задачи о максимальном потоке, а не задачи о потоке минимальной стоимости. Задача

о максимальном потоке без стоимости ребер возникает во многих приложениях, включая поиск компонентов реберной и вершинной связности и паросочетаний в двудольных графах;

- *перемещение по сети разнотипных продуктов.*

  В телекоммуникационной сети каждое сообщение снабжается определенной информацией о его источнике и месте назначения. Каждый узел-адресат должен получать только те сообщения, которые предназначены ему, а не общий объем сообщений от всех отправителей. Эту ситуацию можно смоделировать в виде задачи о многопродуктовом потоке, где разные запросы определяют разные продукты, и нам требуется удовлетворить все запросы, не превысив пропускную способность ребра.

  Если допускаются фрагментированные потоки, задачу о многопродуктовом потоке можно решить методами линейного программирования. К сожалению, задача о нефрагментированном многопродуктовом потоке является NP-полной даже для двух продуктов.

Алгоритмы решения задач о потоках в сети могут быть сложными и требовать значительных усилий для повышения их производительности. Впрочем, имеются хорошие программы, упоминания о которых вы найдете в *подразделах «Реализации»* и *«Примечания»*. Существуют два основных класса алгоритмов вычисления потоков в сети:

- *методы увеличивающих путей.*

  Эти алгоритмы последовательно находят путь от истока к стоку с положительной пропускной способностью и добавляют его к общему потоку. Можно доказать, что поток в сети является оптимальным тогда и только тогда, когда в ней нет увеличивающего пути. Так как каждое добавление пути увеличивает поток, то в итоге должен быть достигнут глобальный максимум. Разница между алгоритмами этого типа заключается в том, как они выбирают увеличивающий путь. Если не проявлять аккуратность, то каждый новый увеличивающий путь может увеличивать общий поток лишь ненамного, вследствие чего поиск максимального потока может занять много времени;

- *методы проталкивания предпотока.*

  Эти алгоритмы проталкивают потоки от одной вершины к другой, первоначально игнорируя ограничивающее условие, что для каждой вершины исходящий поток должен быть равным входящему. Алгоритмы проталкивания предпотока работают быстрее, чем методы увеличивающих путей, поскольку возможно одновременное увеличение нескольких путей. Эти алгоритмы очень популярны, и они реализованы в программах, упоминаемых далее.

### Реализации

Высокопроизводительные программы для вычисления максимальных потоков и потоков минимальной стоимости были разработаны Эндрю Голдбергом (Andrew Goldberg) и его коллегами. Вычисление максимальных потоков осуществляется программами HIPR и PRF (см. [CG94]), причем в большинстве случаев рекомендуется использовать программу HIPR. Для вычисления потоков минимальной стоимости лучшей является

программа CS (см. Gol97]). Все эти программы написаны на языке C и доступны (для некоммерческого использования) на сайте **http://www.avglab.com/andrew/soft.html**.

На первом соревновании DIMACS по реализациям алгоритмов для задач о потоках в сети и о паросочетаниях (см. [JM93]) было показано несколько реализаций и генераторов, которые можно загрузить из каталога **http://dimacs.rutgers.edu/Challenges**. В число этих программ входят написанные на языке C реализация решения задачи о потоках в сети методом проталкивания предпотока и реализация одиннадцати вариантов решения задачи о потоках в сети, включая старые алгоритмы Диница (Dinic) и Карзанова (Karzanov).

#### *Примечания*

Есть замечательные книги, посвященные задачам о потоках в сети и их применению ([AMO93] и [Wil19]), а работа [GT14] представляет краткое, но подробное разъясняющее обозрение. Фундаментальная теорема о минимальном потоке и максимальном разрезе была доказана Фордом (Ford) и Фалкерсоном (Fulkerson) — см. [FF62]. Сложность задачи многопродуктовых потоков (см. [Ita78]) рассматривается среди прочих в работе [Eve11].

Принято считать, что вычисление потока в сети должно занимать время $O(nm)$. Эта цель была, наконец, достигнута Орлином (Orlin) — см. [Orl13]. Историю алгоритмов для решения этой задачи см. в книге [AMO93]. Экспериментальные исследования алгоритмов вычисления потоков минимальной стоимости вы найдете в [GKK74] и [Gol97].

Потоки информации в сети можно моделировать в виде многопродуктовых потоков, принимая во внимание то обстоятельство, что создание дубликатов информации и манипулирование ими может устранить надобность в отдельных путях от истоков к стокам, когда одну и ту же информацию нужно доставить в несколько стоков. Эти идеи используются в области *сетевого кодирования* (см. [YLCZ05]) для достижения теоретических пределов прохождения информационных потоков, установленных теоремой о минимальном потоке и максимальном разрезе.

### Родственные задачи

Линейное программирование (см. *разд. 16.6*), паросочетание (см. *разд. 18.6*), связность (см. *разд. 18.8*).

## 18.10. Рисование графов

**Вход.** Граф $G$.

**Задача.** Начертить граф $G$ таким образом, чтобы точно отобразить его структуру (рис. 18.10).

### Обсуждение

Рисование графов — вполне естественная задача, но по своей природе она не имеет четкого определения. Что такое *хорошо* нарисованный граф? Нам требуется найти способ визуализации графа, который представляет его структуру самым понятным для человека образом. В то же самое время мы хотим, чтобы нарисованный граф удовлетворял определенным эстетическим критериям.

К сожалению, это расплывчатые критерии, по которым невозможно разработать оптимизирующий алгоритм. Более того, можно создать множество сильно отличающихся

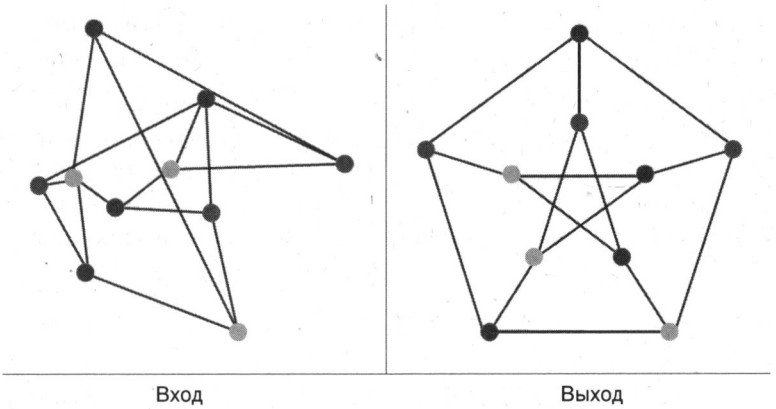



**Рис. 18.10.** Моделирование графа в виде системы пружин

друг от друга представлений одного и того же графа, и каждое из них будет подходящим только для определенного контекста. Например, на рис. 19.10 показаны три разных начертания графа Петерсена (Petersen graph). Какое из них является «правильным»?

Впрочем, существуют и более четкие критерии, позволяющие оценить качество рисунка графа:

- *пересечения.*

  Рисунок должен содержать как можно меньше пересечений ребер, поскольку они отвлекают внимание;

- *площадь.*

  Рисунок должен занимать как можно меньшую площадь относительно длины самого короткого ребра, чтобы избежать нагромождения частей графа;

- *длина ребер.*

  Рисунок не должен содержать длинные ребра, поскольку они могут перекрывать другие элементы рисунка;

- *угловое разрешение.*

  Рисунок не должен содержать очень острых углов между ребрами, инцидентных одной вершине, поскольку это может вызвать частичное или полное наложение линий друг на друга;

- *пропорциональность сторон.*

  Соотношение высоты рисунка к его ширине должно по возможности совпадать с соответствующей характеристикой целевого носителя (как правило, это соотношение сторон экрана компьютерного монитора, равное 4:3), поскольку именно таким образом мы будем в конечном итоге его рассматривать.

К сожалению, эти критерии противоречат друг другу, и задача выбора наилучшего рисунка графа из непустого подмножества таких рисунков предположительно является NP-полной.

Здесь необходимо сделать два замечания. Для графов, не имеющих естественной симметрии или структуры, существование по-настоящему хорошего рисунка, скорее всего, просто невозможно. Это особенно относится к графам, имеющим более 10–15 вершин. Рисунок большого плотного графа сможет отобразить не каждый монитор — даже монитор большого размера. Полный граф из 100 вершин содержит приблизительно 5000 ребер. На мониторе с разрешением 1000×1000 пикселов получается по 200 пикселов на ребро. Что можно увидеть в таком случае?

Тем не менее когда вы со всем этим разберетесь, то увидите, что алгоритмы рисования графов могут быть весьма эффективными. Кроме того, экспериментирование с ними может доставить массу удовольствия. Чтобы выбрать подходящий подход к рисованию вашего графа, попытайтесь найти ответ на следующие вопросы.

- *Должны ли все ребра быть прямыми, или допускаются дуги?*

  Алгоритмы, рисующие прямые линии, сравнительно просты, но имеют свои ограничения. Для визуализации сложных графов — например, электрических схем, лучше всего подходят линии, изломанные под прямыми углами (такие, что все их отрезки расположены или горизонтально, или вертикально — т. е. наклонные линии не разрешаются). При этом каждое ребро графа представлено последовательностью вертикальных и горизонтальных отрезков, соединенных вершинами или точками изгибов.

- *Возможен ли естественный, специфический для приложения рисунок?*

  Если ваш граф представляет дорожную сеть, то вы вряд ли найдете лучший способ нарисовать его иначе, чем поместив все вершины в те же позиции, что и города на карте. Этот фундаментальный принцип действителен для многих других приложений.

- *Является ли граф планарным графом или деревом?*

  Для рисования планарных графов или деревьев воспользуйтесь одним из алгоритмов, описанных в *разд. 18.11* и *18.12*.

- *Является ли граф ориентированным?*

  Ориентация ребра оказывает значительное влияние на характеристики рисунка. При рисовании ориентированных бесконтурных графов лучше всего, чтобы все ребра были направлены в соответствии с некоторым логическим принципом, — например, слева направо или сверху вниз.

- *Насколько быстрым должен быть ваш алгоритм?*

  Если алгоритм предназначается для интерактивного обновления и вывода графов на экран, то он должен работать очень быстро. При этом ваш выбор будет ограничен инкрементальными алгоритмами, которые изменяют положение вершин только в непосредственной близости от любой редактируемой вершины. Если же вы рисуете красивое изображение для длительного изучения, то можете потратить дополнительное время на его оптимизацию.

- *Имеет ли граф оси симметрии?*

  Граф-результат, показанный на рис. 18.10, *справа*, выглядит привлекательно, потому что исходный граф содержит симметрии — в частности, вращательную симметрию в пяти направлениях. Оси симметрии в графе можно определить, найдя его

*автоморфизмы* (изоморфизмы с самим собой). Все автоморфизмы графа можно без труда найти с помощью программ поиска изоморфизмов (см. *разд. 19.9*).

Я рекомендую сначала создать черновой рисунок графа: расположить вершины по кругу на одинаковом расстоянии друг от друга и соединить их прямыми ребрами. Такие рисунки легко поддаются программированию и быстро выполняются. Их значительное преимущество заключается в том, что никакие два ребра не накладываются друг на друга, поскольку в таких рисунках нет трех вершин, расположенных в одну линию. Этих нежелательных эффектов трудно избежать, когда в рисунке разрешается иметь внутренние вершины. Приятным сюрпризом, связанным с круговыми рисунками графов, может оказаться иногда проявляемая ими симметрия, обусловленная тем, что вершины выводятся в том порядке, в котором они определяются в графе. Рисунок можно значительно улучшить, минимизировав длину ребер или количество их пересечений за счет перестановки вершин методом имитации отжига.

Хороший универсальный эвристический алгоритм моделирует граф в виде системы пружин, а при расстановке вершин следует принципу минимизации потенциальной энергии. Пусть смежные вершины притягивают друг друга с усилием, пропорциональным логарифму расстояния между ними, в то время как несмежные вершины отталкивают друг друга с усилием, пропорциональным расстоянию между ними. Эти условия заставляют ребра укорачиваться и в то же самое время вынуждают вершины удаляться друг от друга. Поведение такой системы можно аппроксимировать, определив силу, действующую на каждую вершину в определенный момент времени, а потом переместив каждую вершину на небольшое расстояние в соответствующем направлении. После нескольких таких итераций система стабилизируется на приемлемом рисунке графа. На рис. 18.11 демонстрируется эффективность «встраивания пружин» в конкретный небольшой граф.

Чтобы разработать свой собственный алгоритм рисования графов, вам придется выполнить значительный объем работы, поэтому, если вам нужен, например, алгоритм, рисующий граф ломаными линиями, я рекомендую сначала изучить системы, представленные далее, а также системы, описанные в книге [JM12], и решить, может ли какой-либо из этих алгоритмов подойти для решения вашей задачи.

Задача рисования графов имеет свои «подводные камни», связанные с расстановкой меток у ребер и вершин. Метки нужно помещать близко к ребрам и вершинам, чтобы однозначно идентифицировать их, но в то же время они не должны накладываться ни на элементы графа, ни друг на друга. Можно доказать, что оптимизация размещения меток является NP-полной задачей, но ее можно эффективно решать эвристическими методами, используемыми для решения задачи разложения по контейнерам (см. *разд. 20.9*).

### Реализации

Одной из популярных и хорошо поддерживаемых программ для рисования графов является программа GraphViz (Graph Visualization), разработанная Стивеном Нортом (Stephen North). Подробности см. на сайте **http://www.graphviz.org**. Программа отображает ребра в виде сплайновых кривых и может создавать рисунки весьма больших и сложных графов. В течение многих лет эта программа удовлетворяет все мои профессиональные потребности в рисовании графов.

Все библиотеки структур данных для представления графов, упомянутые в *разд. 15.4*, содержат средства визуализации графов. Разработчики библиотек Boost Graph Library и JGraphT решили не «изобретать колесо», и обе эти библиотеки выводят графы в формате DOT программы GraphViz.

Для интерактивного отображения графов в браузере можно воспользоваться популярными пакетами JavaScript: VivaGraph (**https://github.com/anvaka/VivaGraphJS**) и Sigma (**http: //sigmajs.org/**).

Для задачи рисования графов существуют очень хорошие коммерческие программы — например, программное обеспечение компании Tom Sawyer Software (**www.tomsawyer.com**) и семейство продуктов yFiles (**www.yworks.com**). Пакет Pajek (см. [DNMB18]) специально предназначен для рисования социальных сетей. Загрузить его можно с веб-сайта **http://mrvar.fdv.uni-lj.si/pajek/**. Все эти пакеты предоставляют бесплатные пробные версии или версии для некоммерческого использования.

Библиотека Combinatorica (см. [PS03]) содержит несколько реализаций (на языке пакета Mathematica) алгоритмов рисования графов, включая круговые, пружинные и упорядоченные укладки. Для дополнительной информации по Combinatorica обратитесь к *разд. 22.1.8*.

#### Примечания

Существует многочисленное сообщество исследователей в области рисования графов, которое в течение свыше 25 лет проводит ежегодные конференции Graph Drawing and Network Visualization, посвященные этой теме. Дополнительная информация и протоколы конференции доступны по адресу **http://www.graphdrawing.org/**. Ознакомившись с этими протоколами, вы получите хорошее представление о последних достижениях в области алгоритмов рисования графов и о направлениях, в которых ведутся разработки. Книга [Tam13], является, пожалуй, наиболее полным обзором положения дел в этой области.

Отличными книгами по алгоритмам рисования графов являются [BETT99] и [KW01]. Материал книги [JM12] организован по системам, а не по алгоритмам, однако содержит технические подробности о методах рисования, применяемых в каждой системе. Эвристические алгоритмы для маркировки графов описываются в работах [BDY06] и [WW95].

Укладки графов кодируют структурную информацию о каждой вершине в виде короткого вектора, предоставляя полезные возможности для моделей машинного обучения. Такие двух- или трехмерные укладки можно интерпретировать как позиции вершин для рисования, хотя для машинного обучения более типично измерение размером $d = 128$. Для создания укладок графов большой популярностью пользуется подход с использованием нашего приложения Deepwalk (см. [PARS14]). Обзор этой области представлен в [CPARS18]. Для проецирования многомерных множеств точек (наподобие таких укладок графов) на две размерности для их визуализации широко используется средство t-SNE (см. [MH08]). Реализации этого средства доступны на сайте **https://lvdmaaten.github.io/tsne/**.

Задача равномерного размещения $n$ точек вдоль окружности является тривиальной. Но задача размещения точек на поверхности сферы намного труднее. Для облегчения решения этой задачи были разработаны обширные таблицы для $n \leq 130$ (см. [HSS07]).

### Родственные задачи

Рисование деревьев (см. *разд. 18.11*), проверка на планарность (см. *разд. 18.12*).

# 18.11. Рисование деревьев

**Вход.** Дерево $T$, являющееся графом, не содержащим циклов.

**Задача.** Создать рисунок дерева $T$ (рис. 18.11).

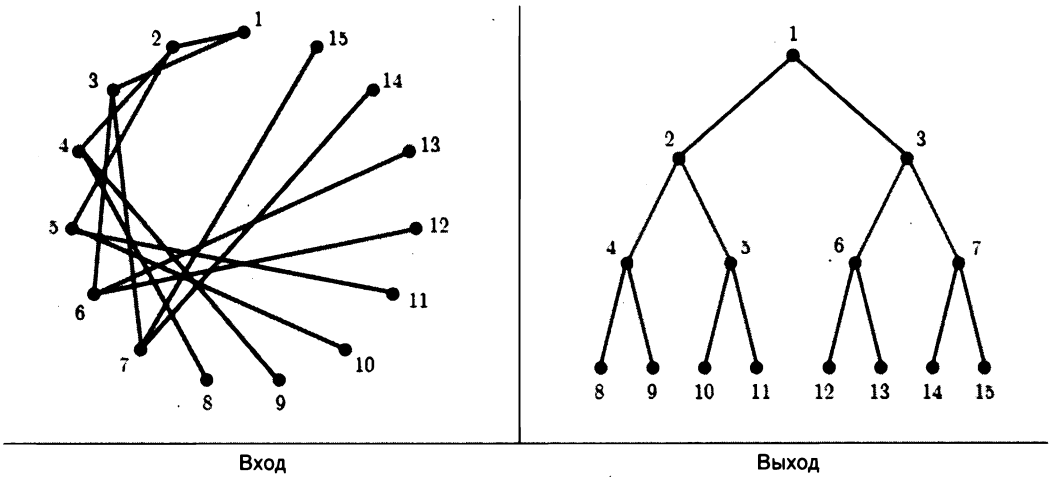



**Рис. 18.11.** Упорядоченная укладка сбалансированного двоичного дерева

### Обсуждение

Во многих приложениях требуется создавать рисунки деревьев. Такие рисунки часто используются для отображения, например, иерархической структуры каталогов файловой системы. Поиск в Google по ключевым словам `tree drawing software` (`программы рисования деревьев`) возвращает около 88 тысяч результатов, содержащих ссылки на специализированные приложения для визуализации генеалогических деревьев, синтаксических деревьев (деревьев грамматического разбора предложения), а также эволюционных филогенетических деревьев.

Каждое приложение предъявляет свои требования к внешнему виду дерева, в результате чего трудно делать какие-либо общие рекомендации. Однако основным вопросом в области рисования деревьев является выяснение, какое дерево нужно нарисовать: свободное или корневое:

- *корневые деревья* определяют иерархически упорядоченную структуру, исходящую из одного узла, называемого *корнем*. Любой рисунок такого дерева должен отражать иерархию его элементов, а также любые специфичные для приложения ограничивающие условия на порядок, в котором следует отображать дочерние узлы. Например, генеалогические деревья имеют корень, а дочерние узлы в них обычно отображаются слева направо, по очередности рождения;
- *свободные деревья* не содержат никакой информации о структуре, кроме топологии соединений. Например, минимальное остовное дерево графа не имеет корня, поэтому иерархическое отображение этого дерева не имеет смысла. Рисунки таких деревьев могут обладать свойствами рисунка полного дерева — например, дорожной

карты, на которой расстояние между населенными пунктами определяет минимальное остовное дерево.

Деревья всегда являются планарными графами, поэтому их можно и нужно рисовать без пересечения ребер. И хотя для этой цели годится любой алгоритм рисования на плоскости из *разд. 18.12*, для создания рисунков деревьев на плоскости можно использовать намного более простые алгоритмы. В частности, эвристические алгоритмы, основанные на «встраивании пружин» (см. *разд. 18.10*), хорошо подходят для рисования свободных деревьев, хотя для интерактивных приложений они могут оказаться слишком медленными.

В большинстве естественных алгоритмов рисования деревьев предполагается, что они работают с корневыми деревьями. Но их можно с таким же успехом применять и для рисования свободных деревьев, выбрав одну из вершин в качестве корневой. Этот псевдокорень можно выбрать произвольно или, что даже лучше, использовать для него *центральную* вершину дерева. Центральная вершина минимизирует максимальное расстояние до других вершин. Этот центр дерева можно найти за линейное время, последовательно удаляя все листья, пока не останется только центральный узел.

При рисовании корневых деревьев вы имеете выбор между *упорядоченной* и *радиальной* укладками.

- *Упорядоченная укладка.*

  Корень помещается вверху по центру страницы, после чего страница разбивается сверху вниз на полосы, количество которых равно степени корня. Удаление корня создает множество поддеревьев, каждое из которых расположено в своей собственной полосе. Каждое поддерево рисуется рекурсивно, причем его новый корень (вершина, смежная со старым корнем) помещается в центре его полосы на фиксированном расстоянии от верха страницы, а старый корень соединяется линией с новым. Упорядоченная укладка сбалансированного двоичного дерева показана на рис. 18.11.

  Такие упорядоченные укладки особенно эффективны для представления какой-либо иерархической структуры, будь то генеалогическое дерево, структура данных или служебная лестница. Расстояние по вертикали показывает удаленность каждого узла от корня. К сожалению, такое последовательное деление на полосы, в конце концов, приводит к появлению очень узких полосок, где большое количество вершин оказываются нагроможденными в небольшой части страницы. Старайтесь отрегулировать ширину каждой полосы так, чтобы отразить общее количество узлов, которые она будет содержать. И не бойтесь занять соседнюю область на полосе после завершения построения более коротких поддеревьев.

- *Радиальная укладка.*

  Свободные деревья лучше рисовать, используя радиальную укладку, в которой центр дерева помещается в центре страницы. Пространство вокруг этой центральной вершины разделяется на секторы для каждого поддерева. Хотя при этом также может возникнуть проблема «скученности», радиальные укладки используют место на странице эффективнее, чем упорядоченные, и являются более естественными для свободных деревьев.

### Реализации

Одной из популярных и хорошо поддерживаемых программ для рисования графов является программа GraphViz (Graph Visualization), разработанная Стивеном Нортом (Stephen North). Подробности см. на сайте **http://www.graphviz.org**. Программа отображает ребра в виде сплайновых кривых и может создавать рисунки весьма больших и сложных графов. В течение многих лет эта программа удовлетворяет все мои профессиональные потребности в рисовании графов. Впрочем, все средства, рассмотренные в *разд. 18.10*, позволяют также выполнять какие-либо разумные операции над деревьями.

Для рисования графов и деревьев существуют очень хорошие коммерческие программы — например, программное обеспечение компании Tom Sawyer Software (**www.tomsawyer.com**) и семейство продуктов yFiles (**www.yworks.com**). В число средств, направленных на работу с рисунками деревьев, можно среди прочих включить также пакеты Lucid (**https://www.lucidchart.com**) и Visme (**https://www.visme.co/tree-diagram-maker/**.) Все эти пакеты предоставляют бесплатные пробные версии или версии для некоммерческого использования.

Библиотека Combinatorica (см. [PS03]) содержит несколько реализаций (на языке пакета Mathematica) алгоритмов рисования деревьев, включая упорядоченные и радиальные укладки. Для дополнительной информации по Combinatorica обратитесь к *разд. 22.1.8*.

#### *Примечания*

Все книги и обзоры по рисованию графов содержат описания алгоритмов, предназначенных для рисования деревьев. Книга [Tam13] является, пожалуй, наиболее полным обзором положения дел в этой области. Отличными книгами по алгоритмам рисования графов являются [BETT99] и [KW01]. Материал книги [JM12] организован по системам, а не по алгоритмам, однако содержит технические подробности о методах рисования, применяемых в каждой системе.

Веб-сайт **https://treevis.net/** содержит обширные ресурсы для визуализации деревьев, а также доклад [Sch11] по связанному исследованию. Популярным методом для отображения иерархических данных является использование *древовидных карт* (treemaps), в которых узлы представляются прямоугольниками, а поддеревья вкладываются в своих родителей. Более подробную информацию см. в [JS91].

Исследования эвристических алгоритмов для укладки деревьев предпринимались многими учеными, а работа [BJL06] отражает последние достижения в этой области. При некоторых ограничениях, носящих эстетический характер, задача рисования графов является NP-полной (см. [SR83]).

Некоторые алгоритмы укладки деревьев получаются из приложений, не связанных с рисованием графов. Метод ван Эмде Боаса организации двоичного дерева обеспечивает более высокую производительность при работе с внешними устройствами хранения данных, чем обычный двоичный поиск, правда, за счет более сложной реализации. Информацию по этой и другим нечувствительным к кэшированию структурам можно найти в [ABF05].

### Родственные задачи

Рисование графов (см. *разд. 18.10*), рисование на плоскости (см. *разд. 18.12*).

# 18.12. Планарность

**Вход.** Граф $G$.

**Задача.** Выяснить, является ли граф $G$ планарным, т. е. возможно ли нарисовать его на плоскости так, чтобы никакие два ребра не пересекались. Если возможно, создать такой рисунок (рис. 18.12).

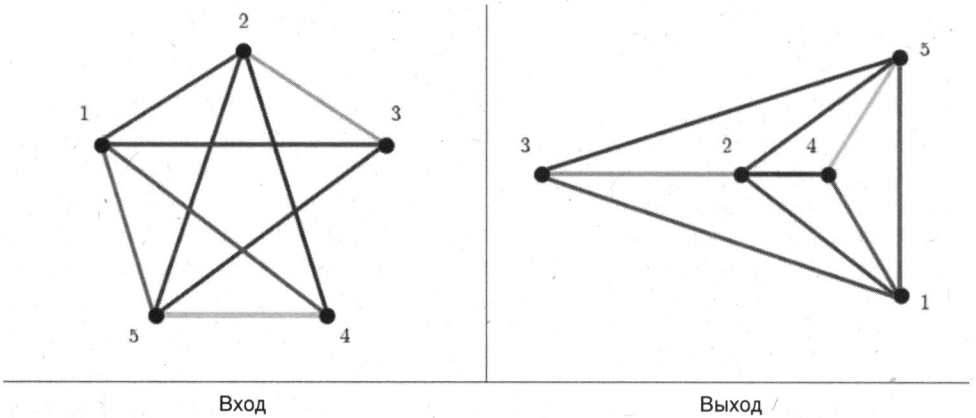



**Рис. 18.12.** Рисунок графа на плоскости

### Обсуждение

Рисование на плоскости (или укладка) графа дает ясное представление о его структуре, поскольку не содержит ни одного пересечения ребер, которое можно было бы принять за вершину. Графы, моделирующие дорожные сети или компоновку печатных плат, являются планарными по своей сути, поскольку полностью определяются плоскостными структурами. По счастливой случайности другие распространенные графы, такие как деревья, также являются планарными.

Планарные графы обладают полезными свойствами, которые можно использовать, чтобы получить более быстрые алгоритмы для решения многих задач. Самый важный факт, который вы должны знать, заключается в том, что все планарные графы являются *разреженными*. Для любого нетривиального планарного графа $G = (V, E)$ справедлива формула Эйлера: $|E| \leq 3|V| - 6$. Это означает, что у каждого планарного графа количество ребер линейно зависит от количества вершин, а также и то, что каждый планарный граф имеет вершину со степенью $\leq 5$. Любой подграф планарного графа является планарным, поэтому всегда должна существовать последовательность вершин низкой степени, которые можно удалить из графа $G$, сводя его в конечном итоге к пустому графу.

Чтобы вы получили более полное представление о тонкостях рисования графов на плоскости, я рекомендую вам создать укладку для графа $K_5 - e$, показанного на рис. 18.12, *слева*. Не подглядывайте на рисунок. Создайте такую укладку, в которой все ребра прямые. Затем добавьте к графу недостающее ребро и попробуйте проделать то же самое для графа $K_5$.

Исследование планарности в значительной степени способствовало развитию теории графов. Однако следует признать, что на практике необходимость в проверке графов

на планарность возникает сравнительно редко. Большинство систем рисования графов не стремится специально создавать планарные укладки. На сайте этой книги Algorithm Repository (**www.algorist.com**) тема выяснения планарности является самой малопосещаемой (см. [Ski99]). Несмотря на это, вам полезно уметь обращаться с планарными графами.

Необходимо понимать разницу между задачей проверки графа на планарность (т. е. выяснением, возможен ли рисунок графа на плоскости) и задачей создания планарной укладки (т. е. собственно созданием рисунка), хотя и то и другое можно выполнить за линейное время. Многие эффективные алгоритмы для работы с планарными графами не используют рисование, а применяют описанную ранее последовательность удаления вершин с низкими степенями.

Алгоритмы проверки на планарность начинают работу с размещения на плоскости произвольного цикла из графа, после чего изучают дополнительные пути в графе, соединяющие вершины с этим циклом. При пересечении двух таких путей один из них нужно рисовать вне цикла, а другой — в цикле. При попарном пересечении трех путей конфликт нельзя разрешить, и, следовательно, граф не может быть планарным. Алгоритмы с линейным временем исполнения, выполняющие проверку на планарность, основаны на обходе в глубину, но они достаточно сложны, так что вам лучше поискать существующую реализацию, а не пытаться создать собственную.

Алгоритмы выявления пересекающихся путей можно использовать для создания планарной укладки, добавляя пути в рисунок по одному. К сожалению, эти алгоритмы работают инкрементально, и ничто не может помешать им поместить слишком много вершин и ребер в небольшую область рисунка. Размещение чрезмерного количества элементов графа в ограниченной области представляет собой большую проблему, поскольку затрудняет восприятие полученных рисунков графов. Поэтому были разработаны более эффективные алгоритмы, создающие *решеточные укладки*, в которых вершины расположены в решетке размером $(2n-4)\times(n-2)$. В результате никакие области рисунка не «захламлены», и ни одно из ребер не оказывается слишком длинным. Тем не менее полученные рисунки обычно выглядят не так естественно, как хотелось бы.

Для непланарных графов часто требуется найти рисунок с минимальным количеством пересечений ребер. К сожалению, задача вычисления количества пересечений ребер графа является NP-полной. Собственно говоря, даже нахождение количества пересечений для планарных графов всего лишь с одним дополнительным ребром является NP-полной задачей (см. [CM13]). Однако существуют эвристические алгоритмы, которые выделяют большой планарный подграф из графа $G$, выполняют укладку этого подграфа, а потом вставляют по одному остальные ребра таким образом, чтобы минимизировать количество пересечений. Этот способ не очень подходит для плотных графов, в которых невозможно избежать большого количества пересечений, но он хорошо работает с почти планарными графами — такими как многослойные печатные платы или сети дорог с эстакадами. Большие планарные подграфы могут быть найдены за счет модифицирования алгоритмов проверки на планарность таким образом, чтобы они удаляли все обнаруженные проблемные ребра.

### Реализации

Библиотека LEDA (см. *разд. 22.1.1*) содержит реализации алгоритмов с линейным временем исполнения как для проверки на планарность, так и для создания решеточных

укладок. Программа проверки планарности этих реализаций возвращает препятствующий подграф Куратовского (см. *подраздел «Примечания»*) для любого предположительно непланарного графа, предоставляя убедительное доказательство его непланарности.

В книге [CGJ+13] представлена среда для рисования графов Open Graph Drawing Framework (**http://www.ogdf.net**), написанная на языке C++. Среда содержит несколько реализаций алгоритмов проверки на планарность и создания планарных укладок, включая реализацию алгоритма для укладки планарных графов на основе PQ-деревьев, рассматриваемого в работе [CNAO85]. Библиотека Boost Graph Library (см. [SLL02]) для языка C++ также содержит алгоритмы для проверки на планарность и создания планарных укладок. Библиотека доступна по адресу **http://www.boost.org/libs/graph**.

Обратите внимание и на PIGALE (**http://pigale.sourceforge.net**) — среду, включающую в себя библиотеку алгоритмов для работы с графами и редактор графов. Она написана на языке C++ и ориентирована в основном на планарные графы. Среда PIGALE содержит разнообразные алгоритмы для создания рисунков графов на плоскости, а также эффективные алгоритмы для проверки на планарность и построения препятствующего подграфа ($K_{3,3}$ или $K_5$), если такой существует.

Процедура GRASP (Greedy Randomized Adaptive Search — «жадный» рандомизированный адаптивный поиск) для поиска наибольшего планарного подграфа реализована Рибейро (Ribeiro) и Ресенде (Resende) (см. [RR99]) и числится в коллекции алгоритмов ACM под номером 797 (см. *разд. 22.1.5*).

#### Примечания

Куратовский в своей работе [Kur30] впервые охарактеризовал планарные графы — показал, что они не содержат подграфа, гомеоморфного графу $K_{3,3}$ или $K_5$. Таким образом, если вы все еще пытаетесь выполнить планарную укладку графа $K_5$, можете прекратить это занятие. Теорема Фари ([F48]) утверждает, что любой планарный граф можно нарисовать так, что все ребра будут прямыми.

Первый алгоритм с линейным временем исполнения для рисования графов был разработан Хопкрофтом (Hopcroft) и Тарьяном (Tarjan) — см. [HT74]. Альтернативный алгоритм проверки на планарность на основе PQ-деревьев разработан Бутом (Booth) и Люкером (Lueker) — см. [BL76]. Упрощенные алгоритмы проверки на планарность описываются в работах [MM96] и [SH99]. Эффективные алгоритмы создания решеточных укладок размером $2n \times n$ впервые были представлены в [dFPP90]. Книга [NR04] содержит хороший обзор алгоритмов рисования планарных графов. Последние обзоры по проверке на планарность излагаются среди прочих в работах [Pat13] и [Tam13].

*Внешнепланарным* называется граф, который можно нарисовать так, чтобы все его вершины принадлежали внешней грани рисунка. Для таких графов характерно отсутствие подграфа, гомеоморфного графу $K_{2,3}$. Распознать их и выполнить их укладку можно за линейное время.

Обобщения планарности графов связаны с укладкой графов в поверхности более сложные, чем плоскость. Я настоятельно рекомендую вам попробовать доказать, что как для $K_{3,3}$, так и для $K_5$ укладку можно выполнить без тороидальных пересечений. Введение в теорию топологических графов дается в [GT01].

### Родственные задачи

Разбиение графов (см. *разд. 19.6*), рисование деревьев (см. *разд. 18.11*).

# NP-сложные задачи на графах

По поводу алгоритмов для работы с графами существует достаточно циничное мнение, что любая задача, возникающая на практике, слишком сложна. Действительно, все задачи в этой главе являются доказуемо NP-полными, за исключением задачи об изоморфизме графов, для которой вопрос сложности остается открытым. Теория NP-полноты гласит, что алгоритм с полиномиальным временем исполнения существует или для всех NP-полных задач, или ни для одной из них. Вероятность первого достаточно незначительная, так что NP-полнота является убедительным доказательством отсутствия эффективного алгоритма решения исходной задачи.

Тем не менее не стоит терять надежду на решение вашей задачи, даже если она рассматривается в этой главе. Для каждой задачи я предлагаю наиболее эффективный подход к ее решению — будь то комбинаторный поиск, эвристический метод, аппроксимирующий алгоритм или алгоритмы для ограниченных экземпляров. Для сложных задач требуется иная методика, чем для задач, решаемых за полиномиальное время, но при правильном подходе с ними можно успешно справиться.

Следующие книги будут весьма полезны вам при решении NP-полных задач:

- [GJ79] — классический справочник по теории NP-полноты. Он содержит краткий каталог свыше 400 NP-полных задач с соответствующими ссылками и комментариями. Обращайтесь к этому каталогу, как только у вас возникнет подозрение в NP-сложности вашей задачи. К этой книге из моей библиотеки по алгоритмам я обращаюсь чаще, чем к остальным;
- [CK97] — доклад, описывающий сборник задач NP-оптимизации, содержащий наилучшие известные аппроксимирующие алгоритмы для каждой из них;
- [WS11] — наиболее полный учебник по теории и разработке аппроксимирующих алгоритмов;
- [Vaz04] — подробное изложение теории аппроксимирующих алгоритмов, написанное авторитетным исследователем этой области;
- [Gon18] — справочник по аппроксимирующим и метаэвристическим алгоритмам, содержит свежие обзоры разных методов решения сложных задач, как прикладных, так и теоретических.

## 19.1. Задача о клике

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти наибольшее подмножество вершин $S$, все пары которого соединены, — т. е. такое, что для всех $(x, y) \in S$ справедливо $(x, y) \in E$ (рис. ЦВ-19.1).

### Обсуждение

Почти каждый из нас может вспомнить, что в его школе существовала «клика» — группа друзей, которые постоянно держались вместе и оказывали влияние на всю социальную жизнь школы. Рассмотрим граф, представляющий школьный социум. Вершины этого графа соответствуют ученикам, а ребра соединяют друзей. Тогда школьная клика будет представлена *кликой* из теории графов — т. е. полным подграфом в графе дружеских отношений.

Задача выявления «кластеров», или родственных объектов, часто сводится к поиску больших клик графа. Интересным примером использования клик является программа, разработанная Налоговой службой США для борьбы с организованным мошенничеством в налоговой сфере. Одно из популярных мошенничеств состоит в том, что в налоговую службу подается большое количество ложных деклараций, составленных в расчете на незаконное получение компенсации. Однако создание большого количества действительно разных деклараций требует больших усилий, и мошенники не всегда делают это корректно. Программа Налоговой службы создает графы, в которых вершины представляют поданные налоговые декларации, и соединяет ребрами заявки, которые выглядят подозрительно похожими. Любая большая клика в графе помогает выявить мошенничество.

Так как клику в общем случае образуют любые две соединенные ребром вершины, трудность заключается не в выявлении хоть какой-нибудь клики, а в поиске именно большой клики. А это уже сложно, поскольку задача поиска максимальной клики является NP-полной. Ситуацию усугубляет тот факт, что сложную задачу представляет собой даже аппроксимация клики с точностью до $n^{1-\varepsilon}$. Теоретически задача поиска клики — это самая сложная задача из рассматриваемых в книге. Итак, на что мы можем надеяться?

Попытайтесь найти ответы на следующие вопросы.

- *Будет ли достаточно максимальной клики?*

  *Максимальной* называется клика, которую нельзя увеличить добавлением вершины. Максимальная клика действительно *может* быть сравнима по величине с наибольшей возможной кликой, но, скорее всего, будет не такой, а намного меньшей. Чтобы найти, будем надеяться, большую максимальную клику, отсортируем вершины по степени в порядке убывания, занесем первую вершину в клику, а потом проверим последующие вершины на смежность со всеми вершинами, добавленными в клику. Если проверяемая вершина удовлетворит этому условию, добавим ее в клику, а если нет, то перейдем к следующей вершине. При использовании битового вектора для пометки вершин, находящихся в клике, этот алгоритм может исполняться за время $O(n + m)$. Альтернативный подход заключается во внесении элемента случайности в упорядочение вершин и принятии в качестве искомой самой большой из максимальных клик, найденных после некоторого количества попыток.

- *Не лучше ли искать большой плотный подграф?*

  Настаивать на выявлениии клик при поиске кластеров рискованно, поскольку нехватка одного ребра может исключить вершину из рассмотрения. Поэтому имеет смысл искать большие *плотные* подграфы — т. е. подмножества вершин, соединен-

ных большим количеством ребер. По определению клики являются наиболее плотными возможными подграфами.

Наибольшее множество вершин, у которых порожденный подграф имеет минимальную вершинную степень $\geq k$, можно найти с помощью простого алгоритма с линейным временем исполнения. Начнем с удаления всех вершин со степенью меньшей чем $k$. Это может понизить степень других вершин до значения, меньшего $k$, поэтому их также нужно удалить. Повторяя этот процесс до тех пор, пока степень всех оставшихся вершин не будет превышать $k$, мы получим наибольший подграф с высокой степенью. Можно создать реализацию этого алгоритма со временем исполнения $O(n + m)$, используя списки смежности и очередь с приоритетами, рассматриваемую в *разд. 15.2*.

- *Как поступать с планарным графом?*

  Планарные графы не могут содержать клики, имеющие больше четырех вершин, в противном случае они не будут планарными. Так как каждое ребро определяет клику из двух вершин, то единственными представляющими интерес кликами в планарном графе являются клики из трех и четырех вершин. Эффективные алгоритмы поиска таких небольших клик рассматривают вершины в порядке возрастания их степени. Любой планарный граф должен содержать вершину $v$ со степенью, не превышающей 5 (см. *разд. 18.12*), поэтому, чтобы найти наибольшую клику, содержащую такую вершину, нужно выполнить полную проверку лишь области постоянного размера. Эта вершина затем удаляется, после чего остается меньший планарный граф, содержащий другую вершину низкой степени. Процесс проверки и удаления продолжается до тех пор, пока граф не станет пустым.

Если вам *действительно* нужно найти самую большую клику графа, то единственное реальное решение — это исчерпывающий перебор с возвратом. Поиск выполняется во всех $k$-подмножествах вершин, и подмножество удаляется, как только в нем обнаруживается вершина, не смежная со всеми остальными вершинами. Очевидной верхней границей размера максимальной клики графа $G$ является наибольшая степень вершины, увеличенная на 1. Более точную верхнюю границу можно получить, отсортировав вершины в порядке убывания их степени. Пусть $j$ будет наибольшим индексом — таким, что степень вершины $v_j$ равна по крайней мере $j - 1$. Самая большая клика графа содержит не больше чем $j$ вершин, поскольку клика размером $j$ не может содержать вершину со степенью, меньшей чем $(j - 1)$. Чтобы ускорить поиск, из графа $G$ сначала следует удалить все такие вершины.

Эвристические алгоритмы поиска больших клик на основе рандомизированных методов — таких как метод имитации отжига, должны работать достаточно хорошо.

### Реализации

Набор процедур Cliquer на языке C, разработанный Патриком Остергардом (Patric Ostergard), предназначен для поиска клик в произвольно взвешенных графах. Процедуры основаны на алгоритме метода ветвей и границ (branch-and-bound algorithm). Загрузить код можно с веб-страницы **http://users.tkk.fi/~pat/cliquer.html**.

Второе соревнование по реализациям DIMACS было посвящено поиску клик и независимых множеств. Наборы данных и код решений с этого соревнования можно загру-

зить по адресу **http://dimacs.rutgers.edu/archive/Challenges/**. Решатель `dfmax.c` реализует простой алгоритм метода ветвей и границ, похожий на алгоритм, описанный в работе [CP90]. А решатель `dmclique.c` использует «полужадный» подход для поиска больших независимых множеств, описанный в работе [JAMS91].

В книге [KS99] рассматриваются программы на языке C на основе метода ветвей и границ для поиска максимальной клики, использующие разные нижние границы. Загрузить эти программы можно с веб-страницы **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Библиотека GOBLIN (**http://goblin2.sourceforge.net/) employs**) содержит реализации алгоритмов метода ветвей и границ для поиска больших клик. Утверждается, что эти реализации могут обрабатывать большие графы, содержащие 150–200 вершин.

#### *Примечания*

Полный обзор задачи поиска максимальных клик приводится в [BBPP99] и [WH15]. Особый интерес представляет работа общества исследования операций по алгоритмам метода ветвей и границ для эффективного поиска клик. Более свежие результаты можно найти в [JS01].

Доказательство NP-полноты задачи о клике было предложено Карпом (Karp) — см. [Kar72]. Выполнив необходимое сведение (см. *разд. 11.3.3*), он установил, что задачи о клике, вершинном покрытии и независимом множестве тесно связаны, поэтому эвристические алгоритмы, которые эффективно решают одну из этих задач, также должны дать удовлетворительные решения для двух других.

В задаче о подграфе с наибольшей плотностью требуется найти такое подмножество вершин, чтобы порождаемый ими подграф имел максимально возможную среднюю степень вершин. Клика из $k$ вершин, очевидно, является самым плотным возможным подграфом этого размера, но менее полные подграфы большего размера могут иметь более высокую степень вершин. Эта задача является NP-полной, но простые эвристические методы на основе последовательного удаления вершин с самой низкой степенью выдают удовлетворительные приблизительные решения (см. [AITT00]). Интересное применение задачи о самом плотном подграфе — обнаружение ссылочного спама в Интернете — описывается в работе [GKT05].

В работе [Has82] было доказано, что клику нельзя аппроксимировать с точностью до $n^{1/2-\varepsilon}$, за исключением того случая, когда $P = NP$ (и с точностью до $n^{1-\varepsilon}$ при более слабых предположениях). Выбор любой отдельной вершины в качестве клики дает аппроксимацию максимальной клики с точностью до $n$. Эти результаты сложности демонстрируют, что никакой аппроксимирующий алгоритм с полиномиальным временем исполнения не может дать сколь-либо лучшие результаты, чем этот простой эвристический алгоритм.

### Родственные задачи

Независимое множество (см. *разд. 19.2*), вершинное покрытие (см. *разд. 19.3*).

## 19.2. Независимое множество

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти такое наибольшее подмножество $S$ множества вершин $V$, в котором нет ребра $(x, y) \in E$, для которого $x \in E$ и $y \in E$ (рис. ЦВ-19.2).

### Обсуждение

Необходимость в поиске большого независимого множества вершин возникает в задаче выбора места для точек обслуживания. Важно не поместить два сервис-центра какого-либо предприятия слишком близко друг к другу, чтобы избежать конкуренции между ними. Для решения этой задачи мы можем создать граф, в котором вершины представляют возможные расположения точек обслуживания, а потом соединить ребрами пары точек, расположенных так близко, что они мешают друг другу. Максимальное независимое множество на этом графе определяет наибольшее количество сервис-центров, которые мы можем расположить без внутренней конкуренции.

В независимых множествах (которые также называются *устойчивыми множествами*) отсутствуют конфликты между элементами, вследствие чего к ним часто прибегают в задачах теории кодирования и календарного планирования. Например, для использования независимого множества в теории кодирования можно определить граф, вершины которого представляют набор возможных кодовых слов, и соединить ребрами любые две вершины, кодовые слова которых похожи друг на друга настолько, что их можно перепутать при шумовых помехах. Максимальное независимое множество этого графа определит код самой большой емкости для рассматриваемого канала связи.

Задача о независимом множестве тесно связана с двумя другими NP-полными задачами:

- *с задачей о клике.*

  Клика — это подмножество вершин, любая пара из которых соединена между собой, тогда как независимым множеством является подмножество вершин, любая пара которых не соединена друг с другом. *Дополнением* графа $G = (V, E)$ является такой граф $G' = (V, E')$, для которого $(i, j) \in E'$ тогда и только тогда, когда $(i, j) \notin E$. Иными словами, в дополнении мы убираем имеющиеся ребра и ставим ребра туда, где их не было. Максимальное независимое множество графа $G$ представляет собой максимальную клику графа $G'$ — следовательно, эти две задачи алгоритмически идентичны. Таким образом, алгоритмы и реализации из *разд. 19.1* можно использовать и для решения задачи о независимом множестве графа $G'$;

- *с задачей вершинной раскраски.*

  Задача вершинной раскраски графа $G = (V, E)$ заключается в разбиении множества вершин $V$ на $k$ подмножеств таким образом, чтобы смежные вершины были разных цветов (см. рис. ЦВ-19.2). Каждый цвет определяет независимое множество. Многие применения задачи о независимом множестве в действительности являются задачами раскраски.

  Один из эвристических подходов к поиску большого независимого множества состоит в применении любого алгоритма вершинной раскраски и использовании самого большого цветового класса. Отсюда следует, что все графы с небольшим количеством цветов (например, планарные или двудольные графы) должны иметь большие независимые множества.

Самый простой эвристический алгоритм поиска независимого множества можно описать так. Находим вершину с самой низкой степенью, добавляем ее в независимое множество, а потом удаляем эту и все смежные с ней вершины. Повторяя процесс, пока

граф не окажется пустым, мы получим *максимальное* независимое множество — т. е. множество, которое нельзя увеличить простым добавлением вершин. Применение рандомизации или исчерпывающего перебора может дать независимые множества чуть большего размера.

Задача о независимом множестве в определенной степени подобна задаче паросочетания в графах. В первой задаче требуется найти большой набор вершин, не имеющих общих ребер, а во второй — большой набор ребер без общих вершин. Это наводит на мысль попробовать переформулировать NP-полную задачу о независимом множестве как задачу паросочетания, которая поддается эффективному решению.

Максимальное независимое множество для дерева можно найти за линейное время следующим способом: удаляются листья, удаленные листья добавляются к независимому множеству, удаляются все смежные узлы, а затем процедура повторяется для полученных деревьев до тех пор, пока дерево не станет пустым.

### Реализации

Любую программу для вычисления максимальной клики в графе можно применить для поиска максимальных независимых множеств, просто дополнив граф. Поэтому все реализации, упомянутые в *разд. 19.1*, подходят и для решения задачи о независимом множестве.

Библиотека GOBLIN (**http://goblin2.sourceforge.net/**) содержит реализацию алгоритма метода ветвей и границ для поиска независимых множеств, которые в инструкции к программе называются *устойчивыми* множествами.

Алгоритм 787 на языке FORTRAN из коллекции алгоритмов ACM, реализованный Ресенде (Resende) — см. [RFS98], представляет собой эвристическую процедуру GRASP для поиска независимого множества (см. *разд. 22.1.5*).

#### *Примечания*

Доказательство NP-полноты задачи о независимом множестве было предоставлено Карпом (Karp) — см. [Kar72]. Эта задача остается NP-полной для планарных кубических графов (см. [GJ79]). Для задачи о независимом множестве на двудольных графах существует эффективное решение (см. [Law11]). Сама задача не является тривиальной, поскольку бóльшая «часть» двудольного графа не обязательно представляет собой максимальное независимое множество.

Решение задачи независимого множества представляет трудность в параллельных и распределенных моделях вычислений, поскольку между выполняющимися одновременно добавлениями вполне могут существовать ребра. Убедительные результаты см. в [BFS12] и [Gha16].

### Родственные задачи

Задача о клике (см. *разд. 19.1*), вершинная раскраска (см. *разд. 19.7*), вершинное покрытие (см. *разд. 19.3*).

# 19.3. Вершинное покрытие

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти наименьшее подмножество $C \subseteq V$, для которого каждое ребро $(x, y) \in E$ содержит по крайней мере одну вершину из этого множества (рис. 19.3).

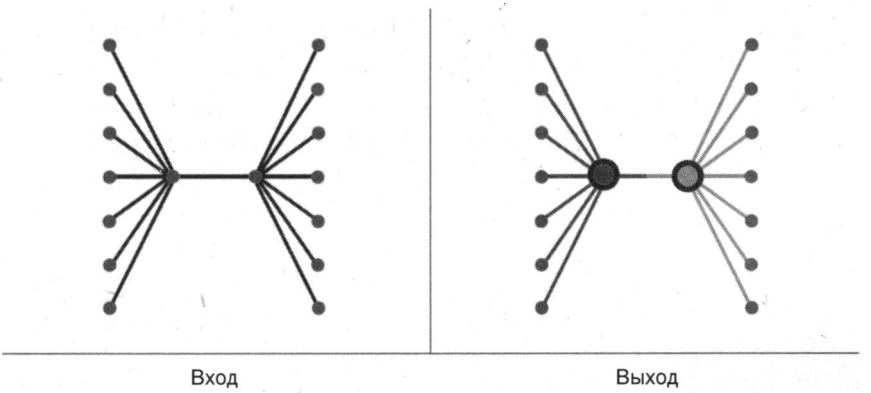



**Рис. 19.3.** Вершинное покрытие

### Обсуждение

Задача о вершинном покрытии является частным случаем более общей задачи о покрытии множества, которая имеет на входе произвольную коллекцию подмножеств $S = (S_1, ..., S_n)$ универсального множества $U = \{1, ..., m\}$ и заключается в поиске наименьшего покрытия $C$ подмножеств из $S$, объединение которых будет равно $U$. Задача о вершинном покрытии часто возникает во многих приложениях, связанных с покупкой товаров, которые продаются в комплектах. Задача о покрытии множества обсуждается в *разд. 21.1*.

Преобразуем задачу о вершинном покрытии в задачу о покрытии множества. Пусть универсальное множество $U$ представляет множество $E$ ребер графа $G$, а множество $S_i$ — набор ребер, инцидентных вершине $i$. В задаче покрытия множества выбор подмножества $S_i$ равнозначен выбору вершины $i$ в задаче вершинного покрытия. Набор вершин определяет вершинное покрытие графа $G$ тогда и только тогда, когда соответствующие подмножества ребер определяют покрытие множества в этом конкретном случае. Так как каждое ребро может находиться только в двух разных подмножествах, то экземпляры задачи о вершинном покрытии проще, чем общая задача покрытия множества. Задача о вершинном покрытии является сравнительно простой среди NP-полных задач и поддается решению более эффективно, чем общая задача о покрытии множества.

Задачи о вершинном покрытии и независимом множестве тесно связаны друг с другом. Так как каждое ребро в множестве $E$ по определению является инцидентным какой-либо вершине в любом покрытии $S$, то невозможно существование ребра, обе вершины которого находились бы в множестве $(V - S)$. Таким образом, множество $(V - S)$ долж-

но быть независимым. Так как минимизация множества $S$ равносильна максимизации множества $(V-S)$, то эти две задачи эквивалентны, и любая программа для решения задачи о независимом множестве также может быть использована и для решения задачи о вершинном покрытии. Наличие двух способов подхода к решению задачи очень удобно, поскольку в некоторых контекстах один из них может оказаться проще.

Самый простой эвристический алгоритм для решения задачи о вершинном покрытии начинает работу с выбора вершины наивысшей степени, добавляет ее к покрытию, удаляет все смежные ребра, а потом повторяет процесс до тех пор, пока граф не станет пустым. На правильно построенных структурах данных этот алгоритм может быть выполнен за линейное время, и в большинстве случаев он выдает «довольно хорошее» покрытие. Но в наихудшем случае для особенно трудных входных экземпляров это покрытие может быть в $\lg n$ раз хуже, чем оптимальное.

К счастью, всегда можно найти вершинное покрытие, размер которого, самое большее, в два раза превышает оптимальный. Для этого находим *максимальное* паросочетание графа $M$ — т. е. множество ребер, не имеющих общей вершины, и которое нельзя расширить, добавляя к нему ребра. Такое максимальное паросочетание можно создать инкрементальным образом: выбираем в графе произвольное ребро $e = (x, y)$, удаляем вершины $x$ и $y$ этого ребра, и повторяем процесс до тех пор, пока в графе не останется больше ребер.

Взяв *обе* вершины каждого ребра в максимальном паросочетании, мы получим вершинное покрытие. Почему это будет вершинным покрытием? Потому что каждое ребро, смежное вершинному покрытию, удаляется. А почему это покрытие должно быть не более чем в два раза большим, чем минимальное покрытие? Потому что *любое* вершинное покрытие должно содержать хотя бы одну из двух вершин каждого ребра паросочетания только для того, чтобы покрыть ребра максимального паросочетания $M$.

Этот эвристический алгоритм можно настроить так, что он будет работать быстрее, — если не в теории, то на практике. Мы можем выбирать ребра паросочетания, чтобы в конечном счете удалить как можно больше других ребер. Начиная с наименьшего максимального паросочетания, которое можно найти, мы минимизируем количество пар вершин в вершинном покрытии. Кроме того, некоторые из вершин из паросочетания $M$ в действительности могут оказаться ненужными для покрытия, поскольку все инцидентные им ребра были «охвачены» другими вершинами паросочетания. Такие избыточные вершины можно идентифицировать и удалить, выполнив второй проход по полученному покрытию.

В задаче о вершинном покрытии требуется охватить все ребра, используя как можно меньшее количество вершин. Есть и две другие задачи, в которых ставятся аналогичные цели:

- *задача о доминирующем множестве.*

  В этой задаче требуется найти такое наименьшее множество вершин $D$, для которого каждая вершина в подмножестве $(V-D)$ соединена ребром по крайней мере с одной вершиной в доминирующем множестве $D$. Каждое вершинное покрытие связного графа также является и доминирующим множеством, но доминирующие множества могут быть намного меньшими, чем вершинные покрытия. Любая от-

дельная вершина представляет минимальное доминирующее множество полного графа $K_n$, в то время как для вершинного покрытия требуется $n - 1$ вершина. Задача о доминирующем множестве обычно возникает в коммуникационных сетях, поскольку оно соответствует множеству узлов, которых достаточно для связи со всеми другими узлами.

Задачи о доминирующем множестве также можно с легкостью представить в виде экземпляров задачи о покрытии множества (см. *разд. 21.1*). Каждая вершина $v_i$ определяет подмножество вершин, в которое входит она сама и все смежные с нею вершины. «Жадный» эвристический алгоритм находит аппроксимацию оптимального доминирующего множества для этого экземпляра за время $\Theta(\lg n)$;

- *задача о реберном покрытии.*

  В этой задаче требуется найти наименьшее множество ребер — такое, что каждая вершина графа инцидентна одному из ребер этого множества. Задачу можно эффективно решить, найдя паросочетание максимальной мощности (см. *разд. 18.6*), а потом выбрав произвольные ребра, чтобы учесть и вершины, не вошедшие в паросочетание. Интересно, что у этих задач-близнецов: реберного и вершинного покрытия — такие разные судьбы: вторая является NP-полной, а первая решается за полиномиальное время.

### Реализации

Любую программу поиска максимальной клики графа можно использовать для решения задачи о вершинном покрытии. Для этого нужно дополнить входной граф и выбрать вершины, которые не принадлежат клике. Поэтому я настоятельно рекомендую вам внимательно изучить программы поиска клики, упомянутые в *разд. 19.1*.

Библиотека графов JGraphT на языке Java (**https://jgrapht.org**) содержит реализации «жадного» и 2-аппроксимирующего эвристических алгоритмов для поиска вершинного покрытия.

#### *Примечания*

Впервые NP-полнота задачи о вершинном покрытии была доказана Карпом (Karp) (см. [Kar72]). Разнообразное множество эвристических методов, включающее рандомизированное округление, составляет основу для 2-аппроксимирующих алгоритмов поиска вершинного покрытия. Хорошие описания этих 2-аппроксимирующих алгоритмов содержатся в работах [CLRS01], [Hoc96], [Pas97], [Vaz04] и [WS11]. Пример «жадного» алгоритма, который может работать хуже оптимального в $\lg n$ раз, впервые приведен в [Joh74] и изложен в книге [PS98]. Экспериментальные исследования эвристических методов для поиска вершинного покрытия описаны в [ACL12], [GMPV06], [GW97] и [RHG07].

Одна из важных нерешенных задач аппроксимирующих алгоритмов касается существования аппроксимирующего решения для задачи о вершинном покрытии, имеющего коэффициент, лучший чем 2. В работе [KR08] представлено доказательство, что, предполагая валидность гипотезы уникальных игр, для задачи вершинного покрытия не существует аппроксимации с коэффициентом $(2 - \varepsilon)$. В работе [DS05] приводится доказательство, что, предполагая $P \neq NP$, для этой задачи не существует аппроксимирующего алгоритма, выдающего решение, имеющее коэффициент, лучший чем 1,36.

Основным справочником по доминирующим множествам является монография [HHS98]. Эвристические методы решения задачи о связном доминирующем множестве рассматриваются в работе [GK98]. Невозможно получить аппроксимирующее решение задачи о доминирующем множестве, имеющее коэффициент, лучший чем $\Omega(\lg n)$ (см. [CK97]]).

### Родственные задачи

Независимое множество (см. *разд. 19.2*), вершинное покрытие (см. *разд. 19.3*).

## 19.4. Задача коммивояжера

**Вход.** Взвешенный граф $G$.

**Задача.** Найти цикл минимальной стоимости, проходящий через каждую вершину графа $G$ только один раз (рис. 19.4).

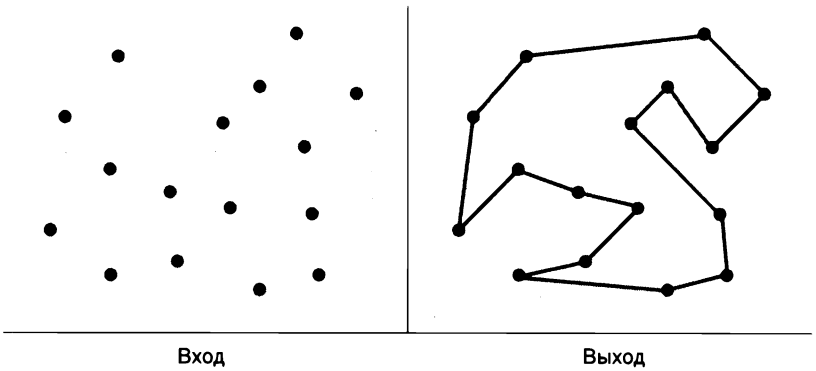



**Рис. 19.4.** Маршрут коммивояжера

### Обсуждение

Задача коммивояжера — наиболее известная из всех NP-полных задач. В качестве причин такой известности можно назвать прикладную ценность задачи и легкость ее популярного изложения. Представьте себе коммивояжера, планирующего поездку на автомобиле в несколько населенных пунктов. Он стремится разработать наиболее короткий маршрут, позволяющий посетить все эти населенные пункты и возвратиться в исходную точку, — т. е. минимизировать общую протяженность поездки.

Задача коммивояжера возникает во многих транспортных приложениях. Оптимизация перемещения инструментов на производстве также является задачей коммивояжера. Возьмем, например, робот-манипулятор для пайки соединений на печатных платах. Наиболее эффективным маршрутом для такого манипулятора станет самый короткий маршрут, посещающий каждую точку пайки.

При решении задачи коммивояжера возникает несколько вопросов.

- *Является ли граф взвешенным?*

  Если граф маршрута является невзвешенным, или полным, графом, в котором вес каждого ребра имеет одно из двух возможных значений (длинное и короткое), то

задача сводится к поиску гамильтонова цикла. Задача поиска гамильтонова цикла обсуждается в *разд. 19.5*.

- *Удовлетворяет ли вход аксиоме треугольника?*

  Наше интуитивное понятие о расстоянии основано на *аксиоме треугольника*. Эта аксиома утверждает, что $d(i, j) \leq d(i, k) + d(k, j)$ для всех вершин $i, j, k \in V$. Геометрические расстояния всегда удовлетворяют аксиоме треугольника, поскольку кратчайший путь между двумя точками проходит по прямой линии между ними. Однако цены на коммерческие авиарейсы не следуют этой аксиоме, что и является причиной трудностей с выбором самого дешевого авиарейса между двумя точками. Эвристические методы решения задачи коммивояжера работают намного лучше с графами разумной степени сложности, удовлетворяющими аксиоме треугольника.

- *Входные данные представляют собой набор из n точек или взвешенный граф?*

  С геометрическими экземплярами часто легче работать, чем с графами, поскольку все пары точек определяют полный граф. При этом поиск подходящего маршрута не вызывает проблем. Вычисляя расстояния между точками лишь по мере надобности, можно также сэкономить память, избавив себя от необходимости хранить матрицу расстояний размером $n \times n$. Геометрические экземпляры по своей сути удовлетворяют аксиоме треугольника, поэтому на них можно достичь производительности, гарантированной некоторыми эвристическими методами. Кроме прочего, можно воспользоваться такими геометрическими структурами данных, как kd-деревья, чтобы быстро найти непосещенные точки, расположенные близко друг к другу.

- *Разрешено ли многократное посещение одной и той же вершины?*

  Для многих приложений запрет на повторное посещение одной и той же вершины не является строгим. Например, при авиаперелетах самый дешевый маршрут облета всех пунктов может проходить через один центральный аэропорт несколько раз. Обратите внимание, что этот вопрос никогда не возникает, когда входной экземпляр удовлетворяет аксиоме треугольника.

  Задача коммивояжера, в которой допускается многократное посещение вершин, легко решается с помощью любой программы, предназначенной для решения общей версии этой задачи, на вход которой подается обновленная матрица стоимостей $D$, у которой ячейка $D(i, j)$ содержит кратчайшее расстояние от точки $i$ к точке $j$. Эта матрица удовлетворяет аксиоме треугольника, и ее можно создать, решив задачу поиска кратчайшего пути между всеми парами точек (см. *разд. 18.4*).

- *Метрика расстояний симметрична?*

  Метрика расстояний *асимметрична*, когда существуют такие точки $(x, y)$, для которых $d(x, y) \neq d(y, x)$. На практике асимметричную задачу коммивояжера решить намного труднее, чем симметричную, поэтому старайтесь избегать таких необычных метрик расстояний. Имейте в виду, что существует способ преобразования асимметричных экземпляров задачи коммивояжера в симметричные, содержащие вдвое больше вершин (см. [GP07]). Вы можете воспользоваться им при решении своей задачи, поскольку программы решения симметричных экземпляров работают намного лучше.

♦ *Насколько важно найти оптимальный маршрут?*

В большинстве случаев достаточно решения, полученного эвристическим методом. Когда на первый план выходит оптимальность, можно применить любой из двух следующих подходов:

- *методы секущих плоскостей* моделируют задачу в виде задачи целочисленного программирования, а потом решают ее соответствующими методами с ослабленными ограничивающими условиями. Если полученное оптимальное решение не является целочисленным, то добавляются дополнительные ограничивающие условия, чтобы обеспечить целочисленность решения;
- алгоритмы метода ветвей и границ выполняют комбинаторный поиск, тщательно соблюдая при этом верхнюю и нижнюю границы стоимости маршрута. При умелом обращении такие алгоритмы могут решать задачи с тысячами вершин. Непрофессионалу для этого придется использовать самую лучшую существующую программу.

Почти любая версия задачи коммивояжера является NP-полной, поэтому правильный подход к ее решению — использование эвристического метода. Такие методы обычно выдают решение, отличающееся от оптимального на несколько процентов, и этого обычно достаточно для практических целей. Для задачи коммивояжера было предложено несколько десятков решений, и выбрать наиболее подходящее не так-то просто. Результаты экспериментов, излагаемые в литературе, несколько противоречивы. Но я рекомендую вам выбрать один из следующих эвристических подходов:

♦ *использование минимальных остовных деревьев.*

Этот эвристический метод начинает с построения минимального остовного дерева, а потом выполняет обход в глубину полученного дерева. При этом каждое из $n-1$ ребер проходится ровно два раза: один раз при открытии вершин во время движения вниз и второй раз на обратном пути вверх. После этого определяется маршрут за счет упорядочивания вершин по времени их открытия. Если граф удовлетворяет аксиоме треугольника, то получившийся маршрут будет самое большее вдвое длиннее оптимального. Но на практике результат обычно еще лучше и превышает оптимальный на 15–20%. Время вычисления минимального остовного дерева для точек на плоскости равно всего лишь $O(n \lg n)$ (см. *разд. 18.3*);

♦ *методы инкрементальной вставки.*

Эвристические алгоритмы этого класса начинают работу с одной вершины, а потом вставляют новые точки в такой частичный маршрут по одной, пока маршрут не будет завершен. Из алгоритмов этого класса лучше всего, по-видимому, работает версия, основанная на вставке *самой дальней точки*: изо всех оставшихся точек в частичный маршрут $T$ добавляется такая точка $v$, для которой

$$\max_{v \in V} \min_{i=1}^{|T|} d\left((v, v_i) + d(v, v_{i+1})\right).$$

Условие min гарантирует, что мы вставляем вершину в позицию, которая добавляет к маршруту наименьшее расстояние, а условие max — что самая «худшая» из таких вершин выбирается первой. Этот подход работает потому, что сначала создается

черновой маршрут и лишь потом уточняются его детали. Полученные таким образом маршруты обычно лишь на 5–10% длиннее оптимальных;

- *использование k-оптимальных маршрутов.*

  Более мощными являются эвристические методы Кернигана — Лина (Kernighan — Lin heuristics). В этих методах сначала выбирается произвольный маршрут, в который потом вносятся локальные изменения с целью его улучшения. В частности, из маршрута удаляются подмножества, состоящие из $k$ ребер, а оставшиеся после этого $k$ цепочек снова соединяются в расчете на получение более короткого маршрута. Маршрут называется $k$-оптимальным, когда из него больше нельзя удалить подмножество из $k$ ребер и по-другому соединить оставшиеся цепочки. Один из эффективных способов улучшения результатов любого другого эвристического алгоритма задачи коммивояжера — выполнить над ним 2-оптимальное улучшение маршрута. В результате экспериментов было установлено, что 3-оптимальные маршруты обычно отличаются от оптимальных лишь на несколько процентов. А для $k > 3$ время вычисления возрастает значительно быстрее, чем качество решения. Метод имитации отжига предоставляет альтернативный механизм обращения ребер для оптимизации решений, полученных эвристическими методами.

### Реализации

Разработанная Эпплгейтом (Applegate), Биксби (Bixby), Чваталом (Chvatal) и Куком (Cook) (см. [ABCC07]) программа Concorde на языке ANSI C предназначена для решения симметричных задач коммивояжера и подобных задач оптимизации сетей. Эта программа установила несколько рекордов и получила оптимальные решения, по крайней мере, для 106 из 110 экземпляров из библиотеки экземпляров задачи коммивояжера (TSPLIB, **http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/**), самый большой из которых содержит 85 900 узлов. Программу Concorde можно загрузить для академических исследований с веб-сайта **http://www.math.uwaterloo.ca/tsp**. Это, бесспорно, самая лучшая реализация решения задачи коммивояжера. На веб-сайте разработчиков можно найти очень интересный материал по истории и практическим приложениям задачи коммивояжера.

Работа [LP07] Лоди (Lodi) и Пуннена (Punnen) — это отличный обзор существующего программного обеспечения для решения задачи коммивояжера. Текущие ссылки на все упоминаемые в этом обзоре программы поддерживаются на веб-сайте **http://or.deis.unibo.it/research_pages/tspsoft.html**.

Библиотека TSPLIB содержит коллекцию стандартных сложных экземпляров задачи коммивояжера, которые возникают на практике. Лучшую версию библиотеки TSPLIB можно загрузить с веб-сайта **http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/**.

#### Примечания

В книге [ABCC07] Эпплгейта (Applegate), Биксби (Bixby), Чвatала (Chvatal) и Кука (Cook) задокументированы методы, использованные авторами в их программах, решающих задачи коммивояжера за рекордное время, а также изложены теоретические основы задачи и ее история. Книга [Coo11] по задаче коммивояжера также пользуется популярностью. Самым лучшим справочником по многочисленным версиям задачи коммивояжера является книга Гутина (Gutin) и Пуннена (Punnen) [GP07], заменившая старый и всеми любимый справочник [LLKS85].

Экспериментальные результаты применения эвристических методов для решения больших экземпляров задачи коммивояжера можно найти в [Ben92a], [WCL+14] и [Rei94]. Эти методы обычно позволяют получить решение, отличающееся от оптимального лишь на несколько процентов.

Эвристический алгоритм Кристофидеса (Christofides) (см. [Chr76]) представляет собой усовершенствованный вариант эвристического метода для построения минимального остовного дерева. На евклидовых графах он гарантирует маршрут, превышающий оптимальный не более чем в полтора раза. Время исполнения этого алгоритма равно $O(n^3)$, а узким местом является поиск совершенного паросочетания с минимальным весом (см. *разд. 18.6*). В результате недавнего решения более общего случая задачи коммивояжера была получена аппроксимация с константным коэффициентом (см. [STV17]). Эвристический алгоритм построения минимального остовного дерева впервые был изложен в работе [RSL77].

Аппроксимирующие методы для евклидовой задачи коммивояжера разработаны Аророй (Arora) и Митчеллом (Mitchell) (см. [Aro98] и [Mit99]). Они выдают приблизительный результат за полиномиальное время с точностью до $1 + \varepsilon$ от оптимального для любого $\varepsilon > 0$. Эти методы представляют большой теоретический интерес, хотя их практическая ценность пока неизвестна.

История поиска оптимальных решений задачи коммивояжера свидетельствует о постоянном прогрессе в этой области. В 1954 году Данциг (Dantzig), Фалкерсон (Fulkerson) и Джонсон (Johnson) решили экземпляр задачи коммивояжера для 42 городов Соединенных Штатов (см. [DFJ54]). В 1980 году Падберг (Padberg) и Хонг (Hong) нашли решение экземпляра, состоящего из 318 вершин (см. [PH80]). А Эпплгейт и др. (см. [ABCC07]) решили экземпляр задачи, размер которого почти в 300 раз больше. Конечно, нельзя сбрасывать со счетов развитие аппаратного обеспечения, но значительная часть успехов объясняется появлением более удачных алгоритмов. Такой темп роста производительности свидетельствует о том, что при острой необходимости получение точных решений для экземпляров NP-полных задач на удивление большого размера вполне возможно.

Для множества точек, образующих выпуклый многоугольник на плоскости, минимальный маршрут коммивояжера определяется их выпуклой оболочкой (см. *разд. 20.2*), которую можно вычислить за время $O(n \lg n)$. Известны также другие простые частные случаи этой задачи.

### Родственные задачи

Гамильтонов цикл (см. *разд. 19.5*), минимальное остовное дерево (см. *разд. 18.3*), выпуклая оболочка (см. *разд. 20.2*).

## 19.5. Гамильтонов цикл

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти маршрут, состоящий из ребер графа, такой что каждая вершина посещается только один раз (рис. 19.5).

### Обсуждение

Задача поиска гамильтонова цикла (или пути) в графе $G$ является частным случаем задачи коммивояжера для графа $G'$ с тем условием, что длина каждого ребра графа $G$ равна 1 в графе $G'$, а расстоянию между вершинами, не соединенными ребрами, при-

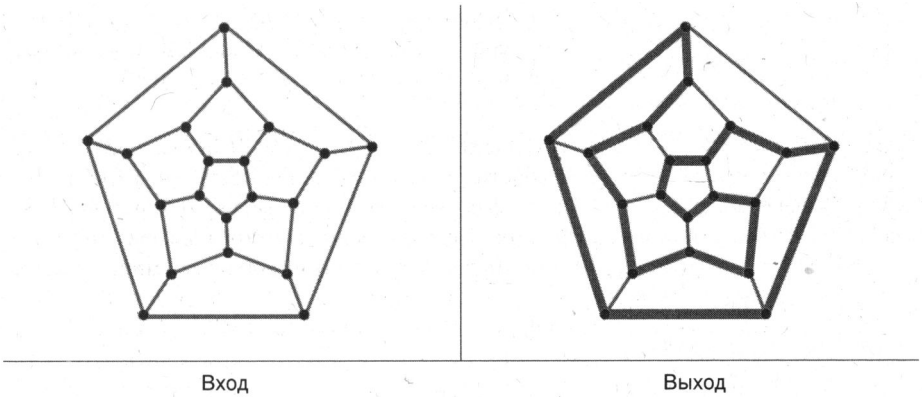



**Рис. 19.5.** Гамильтонов цикл

сваивается большее значение. Такой взвешенный граф содержит маршрут коммивояжера стоимостью $n$ в графе $G'$ тогда и только тогда, когда граф $G$ является гамильтоновым.

Гамильтоновы циклы представляют собой фундаментальные структуры в теории графов и открывают нам полезные способы моделирования разнообразного множества явлений. В *разд. 17.4–17.6* подробно рассматриваются алгоритмы для генерирования комбинаторных объектов: перестановок, подмножеств и разбиений. Наиболее эффективный метод создания таких объектов — это порядок минимального изменения (также называемый *кодами Грея*), который можно естественным образом рассматривать как определение гамильтонова цикла на соответствующем графе.

С задачей поиска гамильтонова пути тесно связана задача поиска самого длинного пути (или цикла) в графах. В задачах календарного планирования с ограничениями на предшествующие вершины, где ориентированное ребро $(x, y)$ подразумевает, что задача $x$ должна быть завершена перед задачей $y$, наиболее длинный путь является критическим путем, который определяет наиболее короткое возможное время для завершения всех задач. А в задачах анализа сигналов в схемах длина наиболее длинного пути (или цикла) определяет время, необходимое для того, чтобы схема приняла стабильное состояние после помехи.

Обе задачи: поиска самого длинного пути и цикла — являются NP-полными даже на невзвешенных графах. Тем не менее существует несколько подходов к их решению, и для выбора наилучшего вы должны найти ответы на следующие вопросы.

- *Назначаются ли крупные штрафы за посещение вершин более одного раза?*

  Если мы переформулируем задачу поиска гамильтонова цикла как минимизацию общего количества вершин, посещенных при полном маршруте, то вместо задачи существования получим задачу оптимизации. Это позволит использовать для ее решения эвристические и аппроксимирующие алгоритмы. Построив остовное дерево графа и выполнив по нему обход в глубину, как показано в *разд. 19.4*, мы получим маршрут, содержащий самое большее $2n$ вершин. Многократное использование рандомизации или метода имитации отжига может значительно уменьшить это число.

- *Является ли бесконтурным ориентированным графом граф, в котором осуществляется поиск самого длинного пути?*

  Задачу поиска самого длинного пути в бесконтурном ориентированном графе можно решить за линейное время с помощью методов динамического программирования. К счастью, для этого можно использовать алгоритм поиска кратчайшего пути в бесконтурном ориентированном графе (представленный в *разд. 18.4*), просто заменив min на max. Бесконтурные ориентированные графы являются наиболее интересными экземплярами задачи поиска самого длинного пути, для которых существуют эффективные алгоритмы.

- *Является ли граф плотным?*

  Достаточно плотные графы всегда содержат гамильтоновы циклы. Более того, циклы на таких графах можно строить достаточно эффективно. В частности, любой граф, все вершины которого имеют степень, равную как минимум $n/2$, должен быть гамильтоновым. Существуют и другие условия достаточности для наличия гамильтоновых циклов — подробности см. в *подразделе «Примечания»*.

- *Нужно ли посетить все вершины или требуется пройти по всем ребрам?*

  Убедитесь в том, что перед вами действительно стоит задача посещения вершин, а не ребер. При должной изобретательности иногда удается переформулировать задачу поиска гамильтонова цикла в терминах задачи поиска эйлерова цикла, в которой требуется пройти по каждому ребру графа. Пожалуй, самым известным таким экземпляром является задача создания последовательностей де Брейна, рассматриваемая в *разд. 18.7*. Выигрыш от такой переформулировки состоит в том, что для решения задачи поиска эйлерова цикла и многих родственных ей задач существуют быстрые алгоритмы, в то время как задача поиска гамильтонова цикла является NP-полной.

Если вам действительно нужно знать, является ли ваш граф гамильтоновым, то единственным решением этой задачи является поиск с возвратом и разрежением поискового пространства. Сначала проверьте граф на двусвязность (см. *разд. 18.8*). Если граф не двусвязный, то это означает, что в нем есть шарнир, удаление которого разъединит граф, и поэтому граф не может быть гамильтоновым.

### Реализации

Описанное ранее сведение (в котором вес ребер равен 1, а расстояние между несмежными вершинами равно 2) преобразовывает задачу поиска гамильтонова цикла в симметричную задачу коммивояжера, удовлетворяющую аксиоме треугольника. Поэтому для решения задачи поиска гамильтонова цикла можно использовать программы дла решения задачи коммивояжера, рассмотренные в *разд. 19.4*. Лучшей из них является программа Concorde, написанная на языке ANSI C и предназначенная для решения симметрических задач коммивояжера и подобных задач оптимизации сетей. Программу Concorde можно загрузить для академических исследований с веб-сайта **http://www.math.uwaterloo.ca/tsp/concorde**. Это, бесспорно, самая лучшая реализация решения задачи коммивояжера.

Эффективная программа для решения задач поиска гамильтонова цикла разработана на основе диссертации Вандергринда (Vandegriend) — см. [Van98]. Код программы и

текст работы можно загрузить с веб-сайта **https://webdocs.cs.ualberta.ca/~joe/Theses/vandegriend.html**.

Лоди (Lodi) и Пуннен (Punnen) написали прекрасный обзор существующего программного обеспечения для решения задачи коммивояжера, включающий частный случай гамильтонова цикла (см. [LP07]). Ссылки на упоминаемые в этом обзоре программы поддерживаются на веб-сайте **http://www.or.deis.unibo.it/research_pages/tspsoft.html**. В работе [NW78] описывается эффективная процедура для перечисления всех гамильтоновых циклов графа методом поиска с возвратом. Подробности вы найдете в *разд. 22.1.9*.

Программа для ранжирования футбольных команд[1] из базы графов Standford GraphBase (см. *разд. 22.1.7*) использует многоуровневый «жадный» алгоритм для решения асимметричной задачи поиска самого длинного пути. Целью задачи является получение цепочки результатов футбольных матчей, чтобы установить превосходство одной футбольной команды над другой. Ведь если команда Университета Вирджинии победила команду Университета Иллинойса с разрывом в 30 очков, а команда Университета Иллинойса победила команду Университета Стоуни Брук с разрывом в 14 очков, тогда, по идее, можно считать, что если бы команда Университета Вирджинии играла с командой Университета Стоуни Брук, то она победила бы ее с разрывом в 44 очка, не так ли? Мы хотим найти самый длинный простой путь в графе, где вес ребра $(x, y)$ обозначает превосходство в счете выигравшей команды $x$ над проигравшей $y$.

#### Примечания

По-видимому, задача поиска гамильтонова цикла впервые возникла в процессе изучения Эйлером задачи о ходе шахматного коня, хотя она и явилась основой изобретенной Гамильтоном головоломки «Вокруг света», опубликованной в 1839 году. Обширный справочный материал по задаче коммивояжера, включающий обсуждение задачи поиска гамильтонова цикла, представлен в [ABCC07], [Coo11], [GP07] и [LLKS85].

Задача вычисления длинных путей в графах представляет большие трудности. Хотя для поиска путей длиной $\Theta(\log n)$ в гамильтоновых графах существуют эффективные алгоритмы (см. [KMR97]), вычислить аппроксимацию даже с полиномиальным коэффициентом трудно (см. [BHK04]). В большинстве учебников по теории графов обсуждаются условия достаточности для наличия гамильтоновых циклов. Моим любимым является учебник [Wes00].

В последнее время большое внимание привлекают к себе методы оптимизации лабораторных биологических процессов. При первом применении этих методов «биовычислений» Аделман (Adleman) (см. [Adl94]) решил задачу поиска гамильтонова пути на экземпляре с семью вершинами. К сожалению, этот подход требует экспоненциального количества молекул, и, зная число Авогадро, можно утверждать, что такие эксперименты невозможны для графов с количеством вершин свыше $n \approx 70$.

### Родственные задачи

Эйлеров цикл (см. *разд. 18.7*), задача коммивояжера (см. *разд. 19.4*).

---

[1] Речь идет об американском футболе. — *Прим. пер.*

# 19.6. Разбиение графов

**Вход.** Граф $G = (V, E)$ (взвешенный) и целые числа $k$ и $m$.

**Задача.** Разбить множество вершин графа на $m$ приблизительно равных подмножеств таким образом, чтобы общая стоимость ребер во всех подмножествах не превышала $k$ (рис. 19.6).

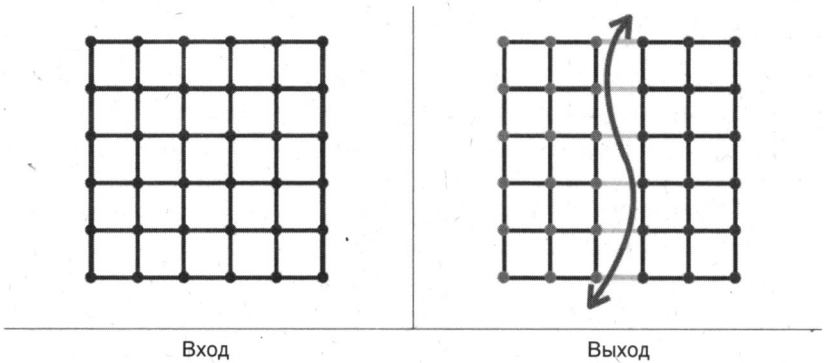



**Рис. 19.6.** Разбиение графа

### Обсуждение

Задача разбиения графа возникает во многих алгоритмах типа «разделяй и властвуй», эффективность которых основана на разделении задачи на несколько меньших подзадач с последующей «сборкой» общего решения из решений этих подзадач. Стремление к минимизации количества ребер, удаленных при разбиении графа, обычно упрощает задачу слияния решений подзадач в одно общее решение.

Задача разбиения графов также возникает при собрании вершин в кластеры логических компонентов. Если ребра соединяют сходные объекты, то оставшиеся после разбиения кластеры должны отражать логически связанные группы. Большие графы часто разбиваются на части приемлемого размера, чтобы облегчить обработку данных или чтобы получить менее загроможденный рисунок.

Наконец, разбиение графов является важным шагом во многих параллельных алгоритмах. В качестве примера можно назвать метод конечных элементов, который используется для расчета физических явлений (таких как механическое напряжение или теплопередача) в геометрических моделях. Параллелизация этих вычислений требует разбиения моделей на равные части с небольшой общей границей. Это тоже задача разбиения графа, поскольку топологии геометрических моделей обычно представляются с помощью графов.

В зависимости от требуемой целевой функции могут возникать разные виды задачи разбиения графа. А именно:

- *минимальный разрез.*

  *Наименьшее* множество ребер, удаление которых разделит граф на части, можно эффективно найти, используя метод потока в сети или рандомизированные алго-

ритмы. Дополнительную информацию по алгоритмам связности см. в *разд. 18.8*. Но такой наименьший разрез может отделить только одну вершину, вследствие чего получившееся разбиение может оказаться очень несбалансированным;

- *разбиение графа.*

  При более практичном критерии разбиения мы ищем разрез небольшого размера, который делит граф на приблизительно равные части. К сожалению, эта версия задачи является NP-полной. Однако на практике она хорошо поддается решению эвристическими методами.

  Некоторые типы графов обязательно имеют небольшие множества вершин-разделителей, которые разбивают граф на сбалансированные части. У любого дерева всегда есть по крайней мере одна вершина $v$, удаление которой (или, эквивалентно, удаление всех смежных с нею ребер) разбивает дерево на части так, что ни одна из них не содержит больше чем половину первоначальных $n$ вершин. Эти компоненты не обязательно являются связными — вспомним, хотя бы, разделяющую вершину звездообразного дерева. Такую разделяющую вершину можно найти за линейное время посредством поиска в глубину.

  Каждый планарный граф имеет множество из $O(\sqrt{n})$ вершин, удаление которых не оставляет компонентов, содержащих более $2n/3$ вершин. Наличие таких разделителей обеспечивает удобный способ разложения геометрических моделей, которые, как правило, часто представляются планарными графами;

- *максимальный разрез.*

  Для графов, представляющих электронные схемы, *максимальный разрез* графа соответствует максимальному потоку данных, который может протекать в схеме. При этом канал связи с самой высокой пропускной способностью должен покрывать разбиение множества вершин, определяемое максимальным разрезом (рис. 19.7). Задача поиска максимального разреза является NP-полной (см. [Kar72]), но на практике хорошо поддается решению эвристическими методами, аналогичными методам разбиения графа.

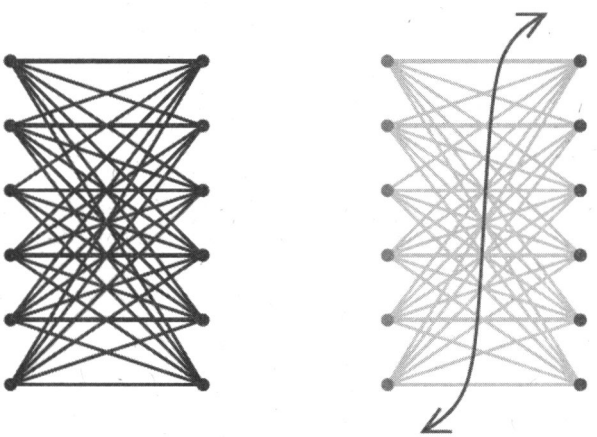


**Рис. 19.7.** Максимальный разрез двудольного графа разрезает все ребра

Основной подход к решению задач разбиения графа и поиска максимального разреза состоит в создании первоначального разбиения множества вершин (либо произвольным образом, либо в соответствии с некоторой стратегией, специфичной для приложения) с последующим перебором всех вершин и принятием решения по поводу каждой из них — не увеличится ли размер разреза, если ее переместить в другую часть графа. Решение о перемещении вершины $v$ можно принять за время, пропорциональное ее степени, выяснив, какая часть разбиения содержит больше ее соседей. Конечно, наиболее подходящая часть для расположения вершины $v$ может измениться после перемещения ее соседей, поэтому, скорее всего, понадобится несколько итераций, прежде чем процесс достигнет локального оптимума. Но даже в этом случае, если нам не повезет, локальный оптимум может быть сколь угодно далек от глобального максимального разреза.

Существует много вариантов этой базовой процедуры, отличающихся от описанного порядком рассмотрения вершин или тем, что в перемещении участвуют целые кластеры вершин. Почти наверняка даст хороший результат использование какого-либо вида рандомизации — особенно метода имитации отжига. Если вершины нужно разбить на более чем две части, то эту процедуру разбиения можно применять рекурсивно.

В *спектральных* методах разбиения используются сложные приемы линейной алгебры. Собственные векторы, связанные с наименьшими ненулевыми собственными значениями *матрицы Лапласа* для графа $G$, предоставляют отличные характеристики для разбиения его на части с высокой степенью связности. Спектральные методы обычно позволяют эффективно находить общую форму разбиения, но результаты можно улучшить, обработав полученный результат методом локальной оптимизации.

### Реализации

Большой популярностью пользуется пакет METIS для разбиения графов, который можно загрузить с веб-сайта **http://glaros.dtc.umn.edu/gkhome/views/metis**. Эта программа успешно применялась для разбиения графов, имеющих более миллиона вершин. Одна из доступных версий программы предназначена для выполнения на многопроцессорных компьютерах, а другая — для разбиения гиперграфов.

Заслуживает внимания и программа Scotch (**http://www.labri.fr/perso/pelegrin/scotch**). Программа Chaco также широко применяется для разбиения графов в приложениях параллельных вычислений. В ней используется несколько разных алгоритмов разбиения, включая метод Кернигана — Лина и спектральный метод. Программу Chaco можно загрузить с веб-сайта **https://cfwebprod.sandia.gov/cfdocs/CompResearch/templates/insert/softwre.cfm?sw=36**.

Десятые соревнования DIMACS (**https://www.cc.gatech.edu/dimacs10/**) были посвящены родственным задачам разбиения и кластеризации графов. Результаты этих соревнований опубликованы в [BMSW13].

#### Примечания

Последние обзоры алгоритмов для разбиения графов предоставляются среди прочих источников в [BS13] и [BMS+16]. Основные эвристические методы для локального улучшения разбиения графа приведены в [KL70] и [FM82]. Спектральные методы разбиения графов рассмотрены в [Chu97] и [PSL90]. Эмпирические результаты по эвристическим методам для разбиения графов изложены в [BG95] и [LR93].

Теорема о планарном разделителе и эффективный алгоритм поиска такого разделителя сформулированы Липтоном и Тарьяном (см. [LT79] и [LT80]). Опыт реализации алгоритмов поиска разделителей планарных графов описан в работах [ADGM07] и [HPS+05].

Можно ожидать, что любое случайное разбиение множества вершин приведет к удалению половины ребер графа, поскольку вероятность того, что две вершины, имеющие общее ребро, окажутся в разных частях разбиения, равна 1/2. Гоманс (Goemans) и Вильямсон (Williamson) в [GW95] представили аппроксимирующий алгоритм, основанный на методе полуопределенного программирования, выдающий решение задачи максимального разреза с точностью до 0,878. Более точный анализ этого алгоритма выполнен Карловым (Karloff) — см. [Kar96].

### Родственные задачи

Реберная и вершинная связности (см. *разд. 18.8*), поток в сети (см. *разд. 18.9*).

## 19.7. Вершинная раскраска

**Вход.** Граф $G = (V, E)$.

**Задача.** Раскрасить множество вершин $V$, используя для этого минимальное количество цветов таким образом, чтобы вершины $i$ и $j$ были разного цвета для всех ребер $(i, j) \in E$ (рис. ЦВ-19.8).

### Обсуждение

Задача вершинной раскраски возникает в приложениях календарного планирования и кластеризации. Классическим приложением раскраски графа является распределение регистров при оптимизации программы в процессе компилирования. Для каждой переменной в каждом фрагменте программы существуют интервалы времени, в течение которых ее значение не должно меняться, — в частности, между ее инициализацией и следующим обращением к ней. Следовательно, нельзя помещать в один и тот же регистр две переменные с пересекающимися интервалами времени. Для оптимального распределения регистров создаем граф, в котором вершины соответствуют переменным, и соединяем ребрами каждые две вершины, у которых переменные имеют пересекающиеся интервалы времени. Раскрасив вершины этого графа, можно обеспечить, что никакие две переменные с вершинами одного цвета не конфликтуют между собой, и, таким образом, им можно выделить один и тот же регистр.

Конечно же, никаких конфликтов никогда не будет, если каждая вершина окрашена в свой цвет. Так как количество регистров процессора ограничено, то мы стремимся выполнить раскраску, используя как можно меньшее количество цветов. Минимальное количество цветов, достаточное для раскраски вершин графа, называется его *хроматическим числом*.

На практике встречаются несколько частных случаев задачи раскраски, представляющих интерес. Для их распознавания постарайтесь найти ответы на следующие вопросы.

- *Можно ли раскрасить граф, используя только два цвета?*

  Важным частным случаем является проверка графа на *двудольность*, означающая, что его можно раскрасить, используя только два разных цвета. Двудольные графы

естественно возникают в таких приложениях, как назначение заданий работникам. Быстрые и простые алгоритмы существуют для задач паросочетания (см. *разд. 18.6*), если на них наложено такое ограничение, как двудольность графа.

Проверка графа $G$ на двудольность не составляет труда. Окрашиваем первую вершину в синий цвет, а потом выполняем обход графа в глубину. При открытии каждой новой, еще не окрашенной вершины, ее нужно окрасить в цвет, противоположный цвету смежной с ней вершины, поскольку окраска в тот же самый цвет вызвала бы конфликт. Граф не может быть двудольным, если он содержит ребро $(x, y)$, обе вершины которого окрашены в один цвет. В противном случае конечная раскраска будет двуцветной и раскрашивание займет время $O(n + m)$. Реализацию этого алгоритма вы найдете в *разд. 7.7.2*.

- *Является ли граф планарным? Все ли вершины графа имеют низкую степень?*

  Знаменитая теорема о четырех красках утверждает, что вершинную раскраску любого планарного графа можно выполнить, используя самое большее четыре разных цвета. Для четырехцветной раскраски планарных графов существуют эффективные алгоритмы, но выяснение, можно ли тот или иной планарный граф раскрасить тремя цветами, является NP-полной задачей.

  Однако для вершинной раскраски любого планарного графа в самое большее шесть цветов существует очень простой алгоритм. Любой планарный граф содержит вершину, степень которой не больше пяти. Удаляем эту вершину $v$ и рекурсивно раскрашиваем остальную часть графа. Поскольку вершина $v$ имеет не более пяти соседей, ее всегда можно раскрасить одним из шести цветов, отличающимся от цвета соседних вершин. Метод работает, поскольку после удаления вершины из планарного графа получается планарный граф, и у него тоже должна быть вершина низкой степени, которую можно будет удалить. Эту же идею можно использовать для раскраски любого графа максимальной степени $\Delta$, задействовав не больше $\Delta + 1$ цветов, за время $O(n\Delta)$.

- *Можно ли привести задачу к задаче реберной раскраски?*

  Некоторые задачи вершинной раскраски можно сформулировать в виде задачи *реберной раскраски*, в которой требуется раскрасить ребра графа $G$ таким образом, чтобы ребра с общей вершиной были окрашены разными цветами. Практическая выгода от такой переформулировки состоит в том, что для решения задачи реберной раскраски существует эффективный алгоритм, который возвращает почти оптимальную раскраску. Алгоритмы реберной раскраски рассматриваются в *разд. 19.8*.

Задача вычисления хроматического числа графа является NP-полной. Если требуется точное решение, придется использовать метод перебора с возвратом, который может оказаться неожиданно эффективным при раскраске некоторых графов. Но задача поиска приблизительного решения, достаточно близкого к оптимальному, остается сложной, поэтому не следует ожидать никаких гарантий.

Наилучшими эвристическими алгоритмами для вершинной раскраски являются инкрементальные. Так же как и в ранее упомянутом алгоритме для планарных графов, раскраска вершин выполняется последовательно, при этом цвета для раскраски текущей вершины выбираются в зависимости от уже использованных цветов для раскраски

смежных вершин. Эти алгоритмы различаются способами выбора следующей вершины и выбора цвета для ее раскраски. Практика подсказывает, что вершины следует вставлять в порядке невозрастания их степеней. Дело в том, что для вершин с высокой степенью существует больше ограничений по выбору цвета, и вставлять их надо как можно раньше, чтобы не потребовался дополнительный цвет. Эвристический алгоритм Брелаза (Brelaz) (см. [Bre79]) динамически выбирает неокрашенную вершину, у которой смежные вершины окрашены в наибольшее количество разных цветов, и окрашивает ее неиспользованным цветом с наименьшим номером.

Полученные инкрементальными методами результаты можно улучшить, задействовав механизм *обмена цветов*. Для этого в раскрашенном графе меняем местами два цвета — например, красные вершины раскрашиваем синим цветом, а синие — красным, сохраняя при этом правильную вершинную раскраску. Теперь возьмем раскрашенный должным образом граф и удалим из него все вершины, кроме красных и синих. Мы можем перекрасить одну или несколько из получившихся компонентов связности, сохраняя при этом правильную раскраску. После такого перекрашивания может оказаться, что какая-либо вершина, ранее смежная как с красными, так и с синими вершинами, является смежной только с синими вершинами, что позволит окрасить ее красным цветом.

Обмен цветов позволяет получить более качественную раскраску за счет увеличения времени обработки и сложности реализации. Алгоритмы, основанные на методе имитации отжига и использующие обмен цветов для перехода из одного состояния в другое, будут, скорее всего, еще более эффективными.

### Реализации

Для раскраски графов есть два полезных интернет-ресурса. Страница раскраски графов (**http://webdocs.cs.ualberta.ca/~joe/Coloring/**) предоставляет обширную библиографию и программное обеспечение для генерирования и решения сложных экземпляров задачи раскраски графов. Страница Майкла Трика (Michael Trick) **https://mat.tepper.cmu.edu/COLOR/color.html** содержит хороший обзор приложений для раскраски графов, аннотированную библиографию, а также коллекцию из более чем 70 экземпляро в задачи раскраски графов, возникающих в таких приложениях, как распределение регистров и проверка печатных плат. Оба веб-сайта также включают реализацию на языке C алгоритма раскрашивания DSATUR.

На втором соревновании по реализации алгоритмов DIMACS в октябре 1993 года (см. [JT96]) оценивались программы решения тесно связанных задач поиска клик и раскраски вершин графов. Программы и данные этого соревнования можно загрузить по адресу **http://dimacs.rutgers.edu/Challenges**.

Как библиотека Boost Graph Library для языка C++ [SLL02] (**http://www.boost.org/libs/graph**), так и библиотека JGraphT для языка Java (**https://jgrapht.org**) включают несколько реализаций «жадного» инкрементального эвристического алгоритма для раскраски вершин графов. Библиотека GOBLIN (**goblin2.sourceforge.net**) содержит реализацию алгоритма метода ветвей и границ для вершинной раскраски.

В книге [NW78] приводятся эффективные реализации на языке FORTRAN алгоритмов вычисления хроматических многочленов и раскраски графов, основанных на методе

перебора с возвратом (подробности см. в *разд. 22.1.9*). Библиотека Combinatorica содержит реализации (на языке пакета Mathematica) алгоритмов для проверки двудольности графов, раскраски с помощью эвристических методов, вычисления хроматических многочленов и вершинной раскраски методом перебора с возвратом (подробности см. в *разд. 22.1.8*).

### *Примечания*

Последние обзоры статей по раскраске графов предоставляются среди прочих источников в работах [GHHP13] и [MT10]. Отличным источником информации по эвристическим методам, включающим результаты экспериментов, является уже не новая книга [SDK83]. Классические эвристические методы для вершинной раскраски приводятся в работах [Bre79], [MMI72] и [Tur88]; дополнительные результаты можно найти в [GH06] и [HDD03].

В своей работе [Wil84] Вильф (Wilf) доказал, что работающий методом перебора с возвратом алгоритм для проверки произвольного графа на то, что его хроматическое число равно $k$, выполняется за *постоянное время*, зависящее от $k$, но не зависящее от $n$. Впрочем, это не так впечатлительно, как кажется, поскольку в действительности очень мало графов являются $k$-раскрашиваемыми. Известно несколько эффективных (но тем не менее имеющих экспоненциальное время исполнения) алгоритмов для вершинной раскраски. Обзор таких алгоритмов представлен в [Woe03].

Работа [Pas03] содержит информацию о доказуемо хороших аппроксимирующих алгоритмах вершинной раскраски. С одной стороны, сложность задачи получения аппроксимирующего решения с полиномиальным коэффициентом была доказана в работе [BGS95]. Но, с другой стороны, эвристические методы гарантируют определенные нетривиальные результаты в зависимости от значений различных параметров. Одним из таких методов является алгоритм Вигдерсона (Wigderson) (см. [Wig83]), предоставляющий аппроксимирующее решение, имеющее коэффициент $n^{1-1/(\chi(G)-1)}$, где $\chi(G)$ является хроматическим числом графа $G$.

Теорема Брука утверждает, что для хроматического числа справедливо $\chi(G) \leq \Delta(G) + 1$, где $\Delta(G)$ — максимальная степень вершин графа $G$. Равенство имеет смысл только для циклов нечетной длины (с хроматическим числом 3) и полных графов.

Самой знаменитой задачей в истории теории графов является *задача четырехцветной раскраски*. Она была впервые поставлена в 1852 году и окончательно решена в 1976 году Аппелем (Appel) и Хакеном (Haken), причем для доказательства правильности решения был использован компьютер. Любой планарный граф можно раскрасить пятью цветами, применяя один из вариантов эвристического метода обмена цветов. Несмотря на существование решения задачи четырехцветной раскраски, задача проверки достаточности трех цветов для раскраски конкретного планарного графа является NP-полной. История задачи четырехцветной раскраски и ее доказательство изложены в книге [SK86]. Эффективный алгоритм для четырехцветной раскраски графа представлен в работе [RSST96] и был недавно формально верифицирован ([Gon08]).

### Родственные задачи

Независимое множество (см. *разд. 19.2*), реберная раскраска (см. *разд. 19.8*).

# 19.8. Реберная раскраска

**Вход.** Граф $G = (V, E)$.

**Задача.** Найти наименьший набор цветов, требуемый для раскраски ребер графа $G$ таким образом, чтобы ни одна пара ребер, имеющих общую вершину, не была окрашена одним цветом (рис. ЦВ-19.9).

### Обсуждение

Задача реберной раскраски графов обычно возникает в приложениях календарного планирования, когда требуется минимизировать количество взаимно не конфликтующих маршрутов, необходимых для выполнения того или иного набора заданий. Рассмотрим для примера ситуацию, когда необходимо составить расписание нескольких деловых встреч продолжительностью в один час, в каждой из которых участвуют два сотрудника. Чтобы избежать возможных конфликтов, все встречи можно было бы запланировать на разное время, но разумнее назначить неконфликтующие встречи на одно и то же время. Для решения этой задачи создадим граф $G$, в котором вершины представляют людей, а ребра соединяют тех, кто должен встретиться. Реберная раскраска этого графа определяет искомое расписание. Разные цвета представляют разные временные периоды, а все встречи одного цвета происходят одновременно.

Национальная футбольная лига решает такую задачу реберной раскраски каждый сезон, чтобы составить расписание игр входящих в нее команд. Пары противостоящих команд определяются по результатам игр предыдущего сезона. Составление расписания игр является задачей реберной раскраски, усложненной вторичными ограничивающими условиями, — такими как необходимость в ответных матчах и обеспечение встречи сильных соперников в понедельник вечером.

Минимальное количество цветов, требуемое для реберной раскраски графа, называется *реберно-хроматическим числом* (edge-chromatic number), или *хроматическим индексом* (chromatic index). Ребра цикла четной длины можно раскрасить двумя цветами, в то время как реберно-хроматическое число циклов нечетной длины равно трем.

С задачей реберной раскраски связана интересная теорема, носящая имя *теоремы Визинга*. Согласно этой теореме для любого графа с максимальной степенью вершин $\Delta$ можно выполнить реберную раскраску, используя самое большее $\Delta + 1$ цвет. Чтобы понять это утверждение, обратите внимание на то, что любая реберная раскраска должна содержать как минимум $\Delta$ цветов, поскольку все ребра, инцидентные одной и той же вершине, должны быть раскрашены разными цветами. Это очень жесткое ограничение.

Доказательство теоремы Визинга является конструктивным, потому что его можно представить в виде алгоритма поиска реберной раскраски из $\Delta + 1$ цветов с временем исполнения $O(nm\Delta)$. Задача выяснения, можно ли использовать для раскраски на один цвет меньше, является NP-полной, и ее решение вряд ли стоит требуемых усилий. Реализация теоремы Визинга указывается далее — в *подразделе «Реализации»*.

Задачу реберной раскраски графа $G$ можно преобразовать в задачу вершинной раскраски *реберного* графа $L(G)$, имеющего вершину графа $L(G)$ для каждого ребра в графе $G$ и ребро графа $L(G)$ тогда и только тогда, когда два ребра графа $G$ инцидентны

одной и той же вершине. Реберный граф можно создать за линейное время и раскрасить, используя любую программу вершинной раскраски.

### Реализации

Библиотека Boost Graph Library для языка C++ [SLL02] (**http://www.boost.org/libs/graph**) содержит реализацию конструктивного доказательства теоремы Визинга, выполненную Мисрой и Грайсом (Misra, Gries) и исполняющуюся за время $O(nm)$. В библиотеке GOBLIN (**http://goblin2.sourceforge.net/**) приведен алгоритм метода ветвей и границ для реберной раскраски.

Список программ для вершинной раскраски, которые можно применить к реберному графу, построенному на основе вашего целевого графа, вы найдете в *разд. 19.7*.

#### *Примечания*

Одна из последних работ по реберной раскраске — книга [SSTF12]. Обзоры теоретических исследований задачи реберной раскраски содержатся в [FW77] и [GT94]. Доказательство, что реберную раскраску любого графа можно выполнить, используя для этого самое большее $\Delta + 1$ цвет, было независимо представлено Визингом (Vizing) (см. [Viz64]) и Гуптой (Gupta) (см. [Gup66]). Простое конструктивное доказательство этого результата приводят Мисра (Misra) и Гриз (Gries) в своей работе [MG92]. Несмотря на ее специфичность, задача вычисления реберно-хроматического числа является NP-полной (см. [Hol81]). Реберную раскраску двудольных графов можно выполнить за полиномиальное время (см. [Sch98]).

Представляя реберные графы в своей работе [Whi32], Уитни (Whitney) показал, что за исключением графов $K_3$ и $K_{1,3}$, любые два связных графа, реберные графы которых изоморфны, являются изоморфными. Интересным упражнением может стать поиск доказательства того, что реберный граф эйлерова графа является как эйлеровым, так и гамильтоновым, в то время как реберный граф гамильтонова графа всегда будет только гамильтоновым.

### Родственные задачи

Вершинная раскраска (см. *разд. 19.7*), календарное планирование (см. *разд. 17.9*).

## 19.9. Изоморфизм графов

**Вход.** Графы $G$ и $H$.

**Задача.** Найти отображение $f$ (или все возможные отображения) множества вершин графа $G$ в множество вершин графа $H$, показывающее, что граф $G$ и граф $H$ идентичны, — т. е. $(x, y)$ является ребром графа $G$ тогда и только тогда, когда $(f(x), f(y))$ является ребром графа $H$ (рис. 19.10).

### Обсуждение

Задача выявления изоморфизма графов заключается в проверке графов на идентичность. Допустим, что требуется выполнить затратные операции над каждым графом из некоторой коллекции. Если мы сможем выяснить, что какие-то графы идентичны друг другу, мы станем игнорировать копии, чтобы не делать двойную работу.

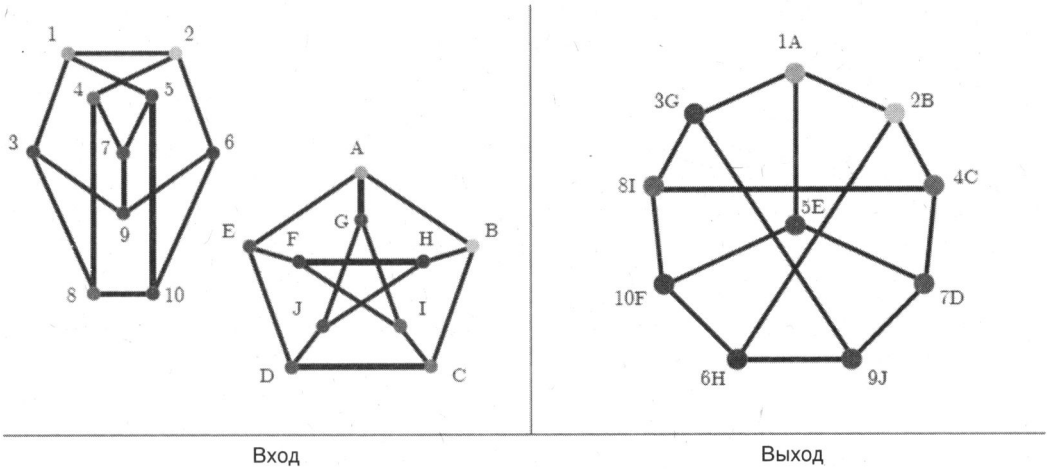



**Рис. 19.10.** Изоморфизм графов

Многие задачи распознавания образов можно сформулировать в виде задачи выявления изоморфизма графов или подграфов. Например, структура химических соединений естественным образом описывается помеченными графами, в которых каждая вершина представляет отдельный атом. Поиск в базе данных всех молекул, содержащих определенную функциональную группу, как раз и является задачей выявления изоморфизма подграфов.

Уточним, что мы понимаем под идентичностью графов. Два помеченных графа $G = (V_g, E_g)$ и $H = (V_h, E_h)$ являются *идентичными*, если $(x, y) \in E_g$ тогда и только тогда, когда $(x, y) \in E_h$. В идентичных графах $v_i$ в графе $G$ соответствует $v_i$ в графе $H$. Более сложная задача выявления изоморфизма заключается в поиске отображения вершин графа $G$ на множество вершин графа $H$, при котором графы идентичны. Задача выявления такого отображения иногда называется *сопоставлением графов*.

Другим важным применением изоморфизма графов является поиск симметрии. Отображение графа на самого себя называется *автоморфизмом*, и коллекция автоморфизмов (*группа* автоморфизмов) содержит много информации о симметричности графа. Например, полный граф $K_n$ содержит $n!$ автоморфизмов (годится любое отображение), в то время как произвольный граф будет иметь, скорее всего, только один автоморфизм, поскольку граф $G$ *всегда* идентичен себе самому. На практике возникает несколько видов задачи изоморфизма графов. Чтобы их распознать, постарайтесь ответить на следующие вопросы.

- *Содержится ли граф G в графе H?*

  Вместо проверки на идентичность мы нередко должны выяснить, является ли тот или иной граф $G$ *подграфом* графа $H$. Такие задачи, как задача о клике, независимом множестве и гамильтоновом цикле, являются важными частными случаями задачи выявления изоморфизма подграфов.

  Выражение «граф $G$ содержится в графе $H$» имеет два разных значения в теории графов. В задаче выявления *изоморфизма подграфа* требуется выяснить, содержит ли граф $H$ подмножество ребер и вершин, которое является изоморфным графу $G$.

А в задаче выявления *изоморфизма порожденного подграфа* требуется выяснить, содержит ли граф $H$ подмножество ребер и вершин, после удаления которого останется подграф, изоморфный графу $G$. В задаче выявления изоморфизма порожденного подграфа требуется, чтобы все ребра графа $G$ присутствовали в графе $H$ и чтобы в графе $H$ не было никаких «не-ребер» графа $G$. Клика представляет собой экземпляр обоих вариантов задачи выявления изоморфизма подграфов, в то время как гамильтонов цикл — это лишь пример простого изоморфизма подграфов.

Следует учитывать это различие при работе над вашим приложением. Задачи выявления изоморфизма подграфов обычно намного сложнее, чем задачи выявления изоморфизма графов, а задачи выявления изоморфизма порожденных подграфов еще сложнее. Единственным разумным подходом к решению таких задач является метод перебора с возвратом.

- *Помечены графы или нет?*

Во многих приложениях вершины и/или ребра графов *помечаются* атрибутами, которые нужно учитывать при выявлении изоморфизма. Например, при сравнении двудольных графов, содержащих вершины двух типов — скажем, «рабочий» и «задание», — любое сопоставление, уравнивающие задание с рабочим, будет лишено смысла.

Метки и связанные с ними ограничивающие условия можно включить в любой алгоритм метода перебора с возвратом. Кроме этого, такие ограничивающие условия могут значительно ускорить поиск, создавая возможности для разрежения пространства поиска при каждом случае несовпадения меток двух вершин.

- *Являются ли деревьями графы, которые проверяются на изоморфизм?*

Для проверки на изоморфизм частных случаев графов, включая деревья и планарные графы, существуют более быстрые алгоритмы. Задача выявления изоморфизма деревьев часто возникает при сопоставлении языковых структур и синтаксическом анализе. Для описания структуры текста используется дерево синтаксического разбора. Два таких дерева: $T_1$ и $T_2$ — будут изоморфными, когда представляемые ими тексты имеют одинаковую структуру.

Эффективные алгоритмы выявления изоморфизма деревьев начинают работу с листьев обоих деревьев и продвигаются к центру. Каждой вершине дерева $T_1$ присваивается метка, представляющая набор вершин во втором дереве $T_2$, который, не исключено, можно сопоставить ему с учетом ограничений, накладываемых метками и степенями вершин. Например, все листья дерева $T_1$ потенциально эквивалентны всем листьям дерева $T_2$. Продвигаясь внутрь, мы можем разбить вершины следующего уровня дерева $T_1$ на классы в зависимости от количества смежных с ними листьев с совпадающими метками. Любое несовпадение означает, что $T_1 \neq T_2$, в то время как по завершении процесса все вершины оказываются разбиты на классы эквивалентности, определяющие все изоморфизмы.

- *Сколько имеется графов?*

Во многих приложениях обработки данных выполняется поиск всех экземпляров графа с определенной структурой в большой базе данных. Уже упоминавшаяся задача отображения химических структур относится к этому типу задач. Такие базы

данных обычно содержат большое количество сравнительно небольших графов. В связи с этим возникает необходимость индексирования базы данных графов по небольшим подструктурам (от пяти до десяти вершин каждая) и выполнения дорогостоящих проверок графов на изоморфность только с теми, которые содержат такие же подструктуры, что и граф запроса.

Для решения задачи выявления изоморфизма графов не предложено ни одного алгоритма с полиномиальным временем исполнения и в то же время не выяснено, является ли эта задача NP-полной. Вместе с задачей разложения на множители целых чисел (см. *разд. 16.8*) это одна из нескольких важных алгоритмических задач, вычислительная сложность которых до сих пор неизвестна даже приблизительно. Принято считать, что если $P \neq NP$, то задача выявления изоморфизма занимает промежуточное положение где-то между P- и NP-полной.

Впрочем, хотя не известно алгоритмов для решения наихудших случаев задачи за полиномиальное время, задача выявления изоморфизма на практике *обычно* не очень сложна. Основной алгоритм выполняет перебор с возвратом всех $n!$ возможных замен меток вершин графа $H$ на метки вершин графа $G$, а потом проверяет графы на идентичность. Конечно же, поиск можно прекратить на определенном префиксе при первом же несовпадении ребер, обе вершины которых имеют этот префикс.

Но настоящим ключом к эффективной проверке на изоморфизм будет предварительное разбиение вершин на два множества так называемых классов эквивалентности таким образом, чтобы две вершины из разных множеств было невозможно никоим образом сопоставить друг другу. Вершины в каждом классе эквивалентности должны иметь одинаковое значение каждого инварианта, не зависящего от меток. Возможно использование одного из следующих инвариантов:

- *степень вершины.*

  Две вершины с разными степенями никогда не могут быть одинаковыми или сопоставленными друг другу. Такое простое разбиение может быть очень полезным, но не в случае с регулярными графами (т. е. графами, у которых все вершины имеют одинаковую степень);

- *матрица кратчайших путей.*

  Матрица кратчайших путей между всеми парами (см. *разд. 18.4*) определяет мультимножество из $n - 1$ расстояний, представляющих расстояния между вершиной $v$ и каждой из остальных вершин. Две вершины могут принадлежать к одному и тому же классу эквивалентности только в том случае, если они определяют одинаковые мультимножества расстояний;

- *количество путей длиной k.*

  Возведение в $k$-ю степень матрицы смежности графа $G$ дает матрицу, в которой ячейка $G^k[i, j]$ содержит количество путей длиной $k$ от вершины $i$ к вершине $j$. Для каждой вершины $v$ и для каждого значения $k$ эта матрица определяет количество путей, которое можно использовать для разбиения вершин на классы, как и расстояния в предыдущем случае.

Используя эти инварианты, часто удается разбить граф на много небольших классов эквивалентности. После такого разбиения вершин не составит труда довести работу до

конца, применив метод перебора с возвратом. Каждой вершине в качестве метки присваивается имя ее класса эквивалентности, и задача решается как задача паросочетания в помеченном графе. Выявить изоморфизм в высокосимметричных графах сложнее, чем в случайных, по причине снижения эффективности таких эвристических методов разбиения вершин на множество классов эквивалентности.

### Реализации

Наиболее известной программой для проверки графов на изоморфизм является программа nauty, представляющая собой набор очень эффективных процедур на языке C для поиска группы автоморфизма графа с вершинной раскраской. Программа также выдает канонически помеченный граф, чтобы содействовать проверке на изоморфизм. Кроме этого, программа может тестировать большинство графов с меньше чем 100 вершинами быстрее, чем за одну секунду. Программу можно загрузить с веб-сайта **http://pallini.di.uniroma1.it/**. Теоретические основы программы nauty, а также аффилированной программы Traces изложены в работах [McK81] и [MP14].

Реализации алгоритмов на языке C++ для проверки изоморфизма графов и подграфов как для деревьев, так и для графов из книги [Val02] доступны для общего пользования. Эти реализации основаны на кодах библиотеки LEDA (см. *разд. 22.1.1*), и загрузить их можно с веб-сайта **http://www.lsi.upc.edu/~valiente/algorithm**.

В книге [KS99] в дополнение к более общим операциям теории графов представлены программы для выявления изоморфизма графов. Эти программы, написанные на языке C, можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

#### *Примечания*

Выявление изоморфизма графов — важная задача теории вычислительной сложности по причине ее редкого статуса задачи с неопределенной сложностью. В 2015 году хорошей новостью в области алгоритмов стало объявление Ласло Бабаи (Lazlo Babai) (см. [Bab16]) — о завершении 40-летних поисков и разработке алгоритма для решения задачи выявления изоморфизма графов с квазиполиномиальным временем исполнения (т. е. меньшим, чем экспоненциальное, но большим, чем полиномиальное).

Среди монографий, посвященных выявлению изоморфизма, можно выделить работу [Hof82] и книгу [KST93]. Книга [Val02] посвящена преимущественно алгоритмам выявления изоморфизма деревьев и подграфов. Подход к проверке на изоморфизм, описанный в книге [KS99], основан на теории групп. Обзор систем и алгоритмов анализа данных, представленных в виде графов, приводится в книге [CH06]. Результаты сравнения производительности разных алгоритмов выявления изоморфизма графов и подграфов содержатся в работе [FSV01].

Существуют алгоритмы с полиномиальным временем исполнения для выявления полиморфизма планарных графов (см. [HW74]) и для графов, у которых максимальная степень вершин ограничена константой (см. [Luk80]). Эвристический метод на основе кратчайшего пути между всеми парами представлен в работе [SD76], хотя существуют неизоморфные графы, имеющие точно такое же множество расстояний (см. [BH90]). Алгоритм с линейным временем исполнения для выявления изоморфизмов в помеченных и непомеченных деревьях приведен в книге [AHU74].

Задача называется *изоморфно-полной*, если она доказуемо столь же сложная, что и задача изоморфизма. Так, задача проверки на изоморфизм двудольных графов является изо-

морфно-полной, поскольку любой граф можно преобразовать в двудольный, заменив каждое из его ребер двумя ребрами, соединенными с новой вершиной. Исходные графы являются изоморфными тогда и только тогда, когда изоморфны графы, полученные в результате такого преобразования.

### Родственные задачи

Задача кратчайшего пути (см. *разд. 18.4*), поиск подстрок (см. *разд. 21.3*).

## 19.10. Дерево Штейнера

**Вход.** Граф $G = (V, E)$, заданное подмножество вершин $T \subseteq V$. Или множество геометрических точек $T$.

**Задача.** Найти наименьшее дерево, соединяющее все вершины $T$ (рис. 19.11).

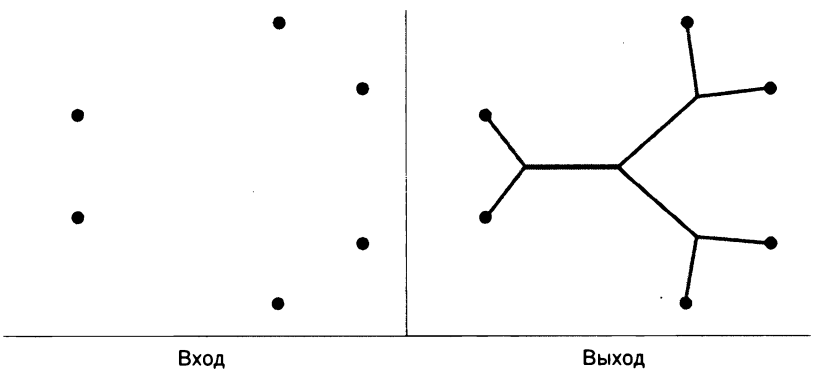



**Рис. 19.11.** Дерево Штейнера

### Обсуждение

Задачи построения деревьев Штейнера возникают при прокладке коммуникационных сетей, поскольку минимальное дерево Штейнера описывает, как соединить имеющееся множество станций, используя кабель наименьшей протяженности. Аналогичные задачи встают перед разработчиками при проектировании водопроводных или вентиляционных и отопительных сетей, а также при создании устройств связи. Типичной задачей построения дерева Штейнера при монтаже электропроводки является требование соединения различных ее точек, например, проводом заземления при соблюдении ограничивающих условий — таких как стоимость материала или задержка распространения сигнала.

Когда $T \neq V$, то задача построения дерева Штейнера отличается от задачи построения минимального остовного дерева в графах (см. *разд. 18.3*), и поэтому нужно выбрать такие промежуточные соединительные узлы, чтобы минимизировать стоимость дерева. Трудная часть геометрической версии задачи построения дерева Штейнера состоит в определении позиций новых точек, которые добавляются для того, чтобы уменьшить длину соединений. Так что прежде чем приступать к построению дерева Штейнера, ответьте на несколько вопросов.

- *Сколько точек требуется соединить?*

  Дерево Штейнера для двух вершин — это просто кратчайший путь между ними (см. *разд. 18.4*). А дерево Штейнера для всех $n$ вершин, когда $T = V$, сводится к определению минимального остовного дерева графа $G$. Несмотря на наличие таких простых частных случаев, общая задача построения минимального дерева Штейнера является NP-полной и остается таковой при широком диапазоне ограничивающих условий.

- *Являются ли входные данные набором точек?*

  Геометрические версии задачи построения дерева Штейнера принимают в качестве входа набор точек, расположенных, как правило, на плоскости, и заключаются в поиске дерева с наименьшим весом, соединяющего эти точки. Набор допустимых промежуточных точек не задается как часть входа, а должен быть получен из исходного набора точек. Эти промежуточные точки должны удовлетворять определенным геометрическим условиям, благодаря которым набор точек-кандидатов становится конечным. Например, в минимальном дереве Штейнера степень каждой точки Штейнера равна 3, а угол между любыми двумя ребрами такой точки должен быть ровно 120°.

- *Существуют ли какие-либо ограничения на определяемые ребра?*

  Во многих задачах монтажа электропроводки все ребра должны располагаться либо горизонтально, либо вертикально. Такая версия задачи называется *линейной задачей Штейнера*. В задаче построения линейного дерева Штейнера на углы между ребрами и степени вершин накладываются иные ограничения, чем в задаче построения евклидова дерева. В частности, все углы должны быть кратны 90°, а максимальная степень любой вершины не должна превышать 4.

- *Действительно ли нужно оптимальное дерево?*

  В некоторых приложениях задачи построения дерева Штейнера большой объем вычислений для построения оптимального дерева Штейнера вполне оправдан. Возможно, например, что разрабатываемая электронная схема будет изготавливаться в миллионах экземпляров, или стоимость прокладки траншей, в которые будут укладываться трубы, составляет тысячи долларов за метр. В таких случаях для построения оптимального дерева следует применять исчерпывающие методы поиска — такие как перебор с возвратом или метод ветвей и границ.

- Существует много возможностей для прореживания пространства поиска с использованием различных геометрических ограничивающих условий. Тем не менее задача построения дерева Штейнера остается сложной. Поэтому, прежде чем пытаться разрабатывать свою собственную реализацию, попробуйте применить уже существующие, описанные далее в *подразделе «Реализации»*.

- *Что означают вершины дерева Штайнера?*

  В приложениях из области классификации и эволюции возникает очень специфичный тип дерева Штейнера. *Филогенетическое дерево* иллюстрирует относительную схожесть разных объектов или биологических видов. Каждый объект обычно представляется листом дерева, а промежуточные вершины соответствуют точкам разветвления между классами объектов. Например, в эволюционном дереве листья мо-

гут представлять *человека*, *собаку*, *змею* и *ящерицу*, а внутренние узлы соответствуют категориям: *животные*, *млекопитающие*, *пресмыкающиеся*. Дерево с корнем *животные*, в котором *человек* и *собака* находятся в категории *млекопитающие*, означает, что человек имеет более близкое родство с собаками, чем со змеями.

Для создания филогенетических деревьев разработано много алгоритмов, которые отличаются друг от друга моделируемыми данными и критериями оптимизации. Разные комбинации реконструирующего алгоритма и метрики выдают разные деревья, поэтому определение «правильного» метода в каждом конкретном случае весьма субъективно. Разумным решением будет применение одного из упомянутых далее стандартных пакетов реализаций и сравнительный анализ результатов обработки данных всеми входящими в него реализациями.

К счастью, существует эффективный эвристический алгоритм для построения деревьев Штейнера, хорошо работающий на всех версиях задачи. Создаем граф, моделирующий вход задачи, и устанавливаем вес ребра $(i, j)$ равным расстоянию от точки $i$ до точки $j$. Минимальное остовное дерево для этого графа гарантированно будет хорошей аппроксимацией как для евклидовых деревьев, так и для линейных деревьев Штейнера.

Самым худшим случаем минимальной аппроксимации остовного дерева евклидова дерева Штейнера является входной экземпляр из трех точек, образующих равносторонний треугольник. В этом случае любое остовное дерево содержит две стороны треугольника (с общей длиной ребер, равной 2), в то время как минимальное дерево Штейнера соединяет все три точки с использованием еще одной стороны — внутренней, и общая длина его ребер равна $\sqrt{3}$. Это соотношение: $\sqrt{3}/2 \approx 0{,}866$ — всегда достижимо, и на практике минимальное остовное дерево обычно имеет размер, на несколько процентов отличающийся от размера оптимального дерева Штейнера. Оптимальное отношение размера линейных деревьев Штейнера к размеру минимального остовного дерева всегда равно $2/3 \approx 0{,}667$.

Для геометрических экземпляров любое минимальное остовное дерево всегда можно улучшить, вставив в него точку Штейнера в любом месте, где угол между инцидентными какой-либо вершине ребрами минимального остовного дерева меньше 120°. Вставив эти точки и откорректировав ребра, можно слегка приблизить решение к оптимальному. Подобная оптимизация возможна и для линейных остовных деревьев.

Заметим, что в этом случае нас интересует только поддерево, соединяющее листья. Минимальное остовное дерево можно разредить, оставив только те ребра, которые лежат на уникальном пути между какой-либо парой конечных узлов. Полный набор таких ребер можно найти за время $O(n)$, удаляя любой лист, не являющийся конечным узлом, а затем повторяя процесс.

Альтернативный эвристический алгоритм для графов основан на вычислении кратчайших путей. Начинаем с дерева, состоящего из кратчайшего пути между двумя конечными узлами. Для каждого оставшегося конечного узла $t$ выбираем кратчайший путь к какой-либо вершине внутри дерева и добавляем этот путь, чтобы подсоединить $t$. Качество этого эвристического алгоритма зависит от порядка вставки конечных узлов и от способа вычисления кратчайших путей, но вероятность получить простое и эффективное решение достаточно высока.

### Реализации

Варм (Warme) и др. разработали пакет GeoSteiner, содержащий программы построения как евклидова, так и линейного дерева Штейнера на плоскости (см. [JWWZ18]). Пакет можно также использовать для решения родственной задачи построения минимальных остовных деревьев в гиперграфах. Утверждается, что с его помощью были получены оптимальные решения для экземпляров, состоящих из 10 000 точек. Это, пожалуй, самый лучший пакет для решения геометрических экземпляров задачи построения дерева Штейнера. Загрузить пакет можно с веб-сайта **http://www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/**.

Одиннадцатое соревнование DIMACS, проводившееся в декабре 2014 года, было посвящено алгоритмам для решения задачи деревьев Штейнера. На нем обсуждались реализации эффективных алгоритмов для вычисления кратчайших путей. Доклады, экземпляры задач и реализации алгоритмов с этого соревнования доступны на сайте **http://dimacs.rutgers.edu/programs/challenge/**.

Пакет FLUTE (**http://home.eng.iastate.edu/~cnchu/flute.html**) предназначен для быстрого построения линейных деревьев Штейнера. Программа предоставляет управляемый пользователем параметр для контроля соотношения между качеством решения и временем исполнения. Библиотека GOBLIN (**http://goblin2.sourceforge.net/**)) содержит реализации как эвристических, так и поисковых алгоритмов для построения деревьев Штейнера в графах.

Пакеты PHYLIP (**http://evolution.genetics.washington.edu/phylip.html**) и PAUP (**https://paup.phylosolutions.com/**) широко применяются для построения филогенетических деревьев и содержат реализации свыше 20 разных алгоритмов для их создания. Хотя многие из них предназначены для работы с моделями молекулярных последовательностей, некоторые методы принимают в качестве входа произвольные матрицы расстояний.

#### *Примечания*

В числе монографий, посвященных деревьям Штейнера, можно назвать [HRW92] и [PS02]. Книга [DSR00] содержит коллекцию обзоров по всем аспектам построения деревьев Штейнера. Результаты исследований эвристических методов работы с деревьями Штейнера изложены в [BC19], [SFG82] и [Vos92].

Задача построения евклидовых деревьев Штейнера впервые была поставлена Ферма, который изучал вопрос поиска на плоскости точки $p$ такой, что сумма расстояний до трех других точек минимальна. Эта задача была решена Торричелли до 1640 году. По-видимому, над общей задачей для $n$ точек работал также Штейнер, и поэтому ему по ошибке приписывается авторство ее постановки. Подробное и интересное изложение истории задачи представлено в работе [HRW92].

Гилберт (Gilbert) и Поллак (Pollak) (см. [GP68]) были первыми, кто выдвинул предположение, что отношение размера минимального дерева Штейнера к размеру минимального остовного дерева всегда $\geq \sqrt{3}/2 \approx 0{,}866$. После двадцати лет активных исследований отношение Гильберта — Поллака было наконец доказано Ду (Du) и Хвангом (Hwang) — см. [DH92]. Евклидово минимальное остовное дерево для $n$ точек на плоскости можно создать за время $O(n \lg n)$ (см. [PS85]).

Аппроксимирующая схема с полиномиальным временем исполнения для деревьев Штейнера в $k$-мерном евклидовом пространстве была представлена в работе [Aro98]. А в работе

[RZ05] приведен метод получения аппроксимирующего решения задачи построения деревьев Штейнера для графов, имеющего коэффициент 1,55.

Сложность задачи построения дерева Штейнера для евклидовых и манхэттенских метрик была доказана в работах [GGJ77] и [GJ77]. Неизвестно, является ли задача построения евклидова дерева Штейнера NP-полной из-за проблем с представлением позиций точек Штейнера.

Можно провести аналогию между минимальными деревьями Штейнера и структурами с минимальной энергией в некоторых физических системах. Предположение, что такие системы (например, мыльные пленки на проволочных рамках) «решают» задачу построения дерева Штейнера, обсуждается в [DKR10]. С задачей построения деревьев Штейнера также довольно хорошо справляются слизевики (см. [LSZ+15]).

### Родственные задачи

Минимальное остовное дерево (см. *разд. 18.3*), поиск кратчайшего пути (см. *разд. 18.4*).

## 19.11. Разрывающее множество ребер или вершин

**Вход.** Ориентированный граф $G = (V, E)$.

**Задача.** Найти наименьший набор ребер $E'$ или вершин $V'$, удаление которого сделает граф ациклическим (рис. 19.12).

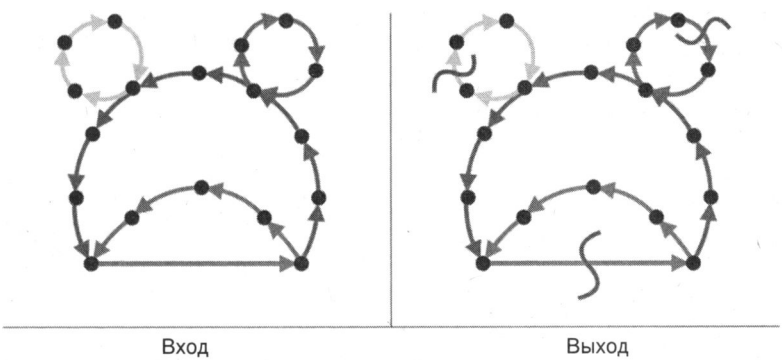



**Рис. 19.12.** Разрывающее множество ребер

### Обсуждение

Необходимость в поиске разрывающего множества возникает из-за того, что многие задачи легче поддаются решению на бесконтурных ориентированных графах, чем на ориентированных графах общего вида. Рассмотрим, например, задачу планирования расписания работ с ограничивающими условиями очередности, требующими, что задача $A$ должна выполняться перед задачей $B$. Когда все ограничивающие условия не противоречат друг другу, то получается бесконтурный ориентированный граф, и для соблюдения условий вершины/работы можно упорядочить посредством топологической сортировки (см. *разд. 18.2*). Но такое расписание составить невозможно, когда существуют циклические ограничивающие условия, — например, задание $A$ нужно выполнить

перед заданием *B*, которое нужно выполнить перед заданием *C*, которое нужно выполнить перед заданием *A*.

Разрывающее множество определяет небольшое количество ограничивающих условий, которые нужно отбросить, чтобы получить допустимое расписание. В задаче *разрывающего множества ребер* (или дуг) мы отбрасываем отдельные ограничивающие условия очередности. А в задаче *разрывающего множества вершин* мы отбрасываем целые задания вместе со связанными с ними ограничениями.

Аналогичным образом устраняется состояние «гонок» в электронных схемах. Задача разрывающего множества также более логично называется задачей поиска *максимального ациклического подграфа*.

Еще одно приложение связано с определением рейтинга спортсменов. Допустим, нам требуется определить рейтинг участников шахматных или теннисных соревнований. Для этого мы можем создать ориентированный граф, в котором ребро идет от вершины *x* к вершине *y*, когда игрок *x* побеждает игрока *y*. По идее, игрок более высокого класса *обычно* должен победить игрока низшего класса, хотя противоположный (неожиданный) результат также наблюдается. Естественным определением рейтинга будет топологическое упорядочение, полученное после удаления из графа минимального разрывающего множества ребер, представляющих неожиданные результаты матча.

Решая задачу разрывающего множества, ответьте на следующие вопросы.

- *Нужно ли отбросить какие-либо ограничивающие условия?*

  Если граф уже является бесконтурным ориентированным графом, что можно выяснить с помощью топологической сортировки, то никакие изменения не требуются. Один из подходов к поиску разрывающего множества состоит в модифицировании алгоритма топологической сортировки таким образом, чтобы при обнаружении конфликта удалялось проблемное ребро или вершина. Но это разрывающее множество может оказаться намного больше оптимального, поскольку на ориентированных графах задачи поиска разрывающего множества ребер и разрывающего множества вершин являются NP-полными.

- *Как найти подходящее разрывающее множество ребер?*

  Эффективный эвристический алгоритм с линейным временем исполнения создает вершинное упорядочение, а потом удаляет каждую дугу, идущую в неправильном направлении. При любом упорядочении вершин направление по крайней мере половины дуг должно быть одинаковым (слева направо или справа налево), поэтому в качестве разрывающего множества следует взять то, которое меньше.

  Как правильно выбрать начальную вершину? Хороший эвристический подход при этом — отсортировать вершины по их реберной разбалансированности, т. е. по разности между степенью захода и степенью исхода. А при подходе с использованием инкрементальных вставок в качестве начальной берется случайная вершина *v*. Любая вершина *x*, определяемая входящим ребром $(x, v)$, будет помещена слева от вершины *v*. Аналогичным образом любая вершина *x*, определяемая исходящим ребром $(v, x)$, будет помещена справа от вершины *v*. Теперь можно рекурсивно выполнить эту процедуру на левом и правом подмножествах, чтобы завершить упорядочение вершин.

- *Как найти хорошее подходящее множество вершин?*

  Только что описанная эвристическая процедура возвращает упорядоченный набор вершин с некоторым количеством обратных ребер. Нам нужно найти небольшое множество вершин, покрывающее эти обратные ребра. То есть мы имеем задачу о вершинном покрытии, эвристические методы решения которой рассматриваются в *разд. 19.3*.

- *Как разорвать все циклы в неориентированном графе?*

  Задача поиска разрывающих множеств в неориентированных графах существенно отличается от задачи их поиска в ориентированных. Деревья являются ациклическими неориентированными графами, и каждое дерево из $n$ вершин содержит ровно $n-1$ ребро. Таким образом, мощность наименьшего разрывающего множества ребер любого неориентированного графа $G$ будет $|E|-(n-c)$, где $c$ — количество компонентов связности графа $G$. Обратные ребра, обнаруженные при обходе в глубину графа $G$, можно рассматривать как минимальное разрывающее множество ребер.

  Задача поиска разрывающего множества вершин для неориентированных графов является NP-полной. Эвристическая процедура ее решения использует обход в глубину для поиска самого короткого цикла графа. Из графа $G$ удаляется одна из вершин найденного цикла (или все вершины, если нужен гарантированный результат), и в оставшемся графе вновь выполняется поиск оставшегося цикла минимальной длины. Эта процедура поиска и удаления циклов повторяется до тех пор, пока граф не станет ациклическим. Оптимальное разрывающее множество вершин должно содержать хотя бы одну вершину, принадлежащую каждому из этих циклов (у которых нет общих вершин), и поэтому качество полученного приблизительного решения определяется средней длиной удаленных циклов.

Иногда имеет смысл улучшить эвристические решения, используя рандомизацию или метод имитации отжига. Для перехода из одного состояния в другое можно изменять порядок вершин, обменивая пары вершин местами или вставляя/удаляя вершины множества, являющегося кандидатом на роль разрывающего.

### Реализации

Алгоритм 815 из коллекции алгоритмов ACM (см. *разд. 22.1.5*) представляет собой эвристическую процедуру GRASP для поиска разрывающего множества как вершин, так и ребер (см. [FPR01]).

В соревновании по параметризированным алгоритмам и вычислительным экспериментам (Parameterized Algorithms and Computational Experiments Challenge) — см. **https://pacechallenge.wordpress.com/track-b-feedback-vertex-set/** — первое место занял точный решатель задачи поиска разрывающего множества вершин авторства Йоичи Ивата (Yoichi Iwata) и Кенсуке Иманиши (Kensuke Imanishi) — см. [Iwa16].

Библиотека GOBLIN (**http://goblin2.sourceforge.net**) содержит реализацию аппроксимирующего алгоритм поиска минимального разрывающего множества дуг.

Программа econ_order из базы графов Standford GraphBase (см. *разд. 22.17*) выполняет перестановку рядов и столбцов матрицы таким образом, чтобы минимизировать суммы

чисел под главной диагональю. Использование матрицы смежности в качестве входа и удаление всех ребер под главной диагональю позволяет получить ациклический граф.

### Примечания

Обзор решений задачи разрывающего множества представлен в [FPR99]. Обсуждение доказательства сложности задачи поиска минимального разрывающего множества (см. [Kar72]) можно найти в [AHU74] и [Eve11]. Задачи разрывающего множества вершин и задачи разрывающего множества ребер остаются сложными даже в том случае, когда степень захода и исхода всех вершин не больше двух (см. [GJ79]).

В работе [BBF99] представлен аппроксимирующий алгоритм, выдающий решение для задачи разрывающего множества вершин, имеющее коэффициент 2. Разрывающее множество ребер в ориентированных графах можно аппроксимировать с коэффициентом $O(\log n \log \log n)$ (см. [ENSS98]). В работе [LMM+18] рассматривается эвристическая процедура определения рейтинга участников соревнований. Эксперименты с эвристическими методами описаны в журнале [Koe05].

Управляемые алгоритмы с фиксированными параметрами (fixed-parameter tractable algorithms) являются полиномиальными по размеру входных данных $n$ и экспоненциальными по размеру решения $k$ — скажем, $O(k!n)$. При условии константного $k$ такие алгоритмы будут линейными. Эти алгоритмы можно использовать для решения задачи вычисления разрывающего множества вершин. См. [CCL15]. Наилучший обзор сложности алгоритмов с фиксированными параметрами представлен в книге [DF12].

Я должен признаться, что применяю подход с использованием разрывающего множества ребер для оценки проектов в моем аспирантском курсе по теории и анализу данных, поскольку не могу оценить все работы из-за громадного их количества. Ассистентам преподавателя и студентам, как правило, можно доверять бинарное суждение о том, лучше или хуже работа $x$, чем работа $y$. Получив большое количество таких оценочных пар, я удаляю наименьшее количество конфликтных оценок при помощи алгоритма вычисления разрывающего множества ребер, а затем упорядочиваю работы с помощью топологической сортировки, чтобы поставить им оценки.

Интересное применение задачи разрывающего множества дуг в экономике представлено в книге [Knu94]. Для каждой пары $A$ и $B$ секторов экономики дается информация об объеме потока финансов из сектора $A$ в сектор $B$. Секторы упорядочиваются таким образом, чтобы выяснить, какие из них являются преимущественно поставщиками для других секторов, а какие поставляют товар в основном потребителям.

## Родственные задачи

Уменьшение ширины ленты матрицы (см. *разд. 16.2*), топологическая сортировка (см. *разд. 18.2*), календарное планирование (см. *разд. 17.9*).

# Вычислительная геометрия

Вычислительная геометрия — это область дискретной математики, в которой изучаются алгоритмы решения геометрических задач. Ее развитие совпало с возникновением таких прикладных областей, как компьютерная графика и системы автоматизированного проектирования и производства, а также с более широким применением компьютеров в различных научных отраслях. Во всех этих областях требуется применение вычислительной геометрии.

Среди книг по вычислительной геометрии можно выделить следующие:

- [dBvKOS08] — самое лучшее введение в общую теорию вычислительной геометрии и ее основные алгоритмы;
- [O'R01] — прекрасное практическое введение в вычислительную геометрию. В нем особое внимание уделяется тщательной и правильной реализации алгоритмов. Код на языках C и Java можно загрузить с веб-страницы **https://cs.smith.edu/~orourke/books/compgeom.html**;
- [PS85] — несмотря на солидный возраст, эта книга остается хорошим введением в вычислительную геометрию. В ней подробно описаны алгоритмы построения выпуклой оболочки, построения диаграмм Вороного и выявления пересечений;
- [TOG18] — недавно изданный сборник статей содержит подробный обзор положения дел почти во всех областях дискретной и вычислительной геометрии.

Симпозиум ассоциации ACM по вычислительной геометрии (ACM Symposium on Computational Geometry) проводится ежегодно в последних числах мая или в первых числах июня. Существует постоянно расширяющаяся база реализаций алгоритмов вычислительной геометрии. Как правило, в этой главе указываются конкретные реализации, но вам определенно следует обратить особое внимание на обширную библиотеку CGAL (Computational Geometry Algorithms Library — библиотека алгоритмов вычислительной геометрии). Она содержит реализации большого количества алгоритмов на языке C++, разработанные как часть крупного общеевропейского проекта. Каждому, кто серьезно интересуется вычислительной геометрией, следует ознакомиться с этой библиотекой (**http://www.cgal.org**).

## 20.1. Элементарные задачи вычислительной геометрии

**Вход.** Точка $p$ и отрезок $l$ или два отрезка $l_1$ и $l_2$.

**Задача.** Выяснить, по какую сторону от отрезка $l$ расположена точка $p$. Выяснить, пересекаются ли отрезки $l_1$ и $l_2$ (рис. 20.1).

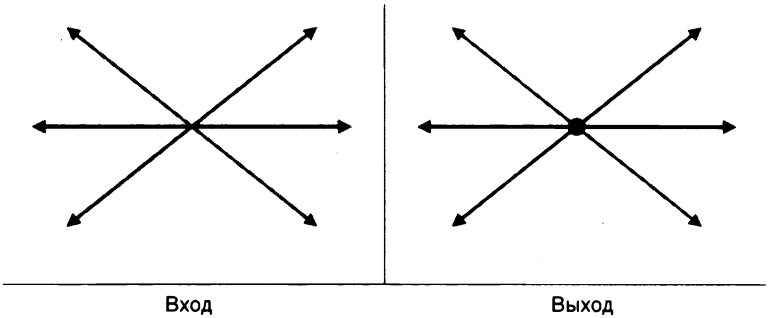



**Рис. 20.1.** Пересечение прямых

### Обсуждение

Элементарные задачи вычислительной геометрии имеют свои «подводные камни». Даже такие простые задачи, как, например, поиск точки пересечения двух прямых, оказываются сложнее, чем кажутся на первый взгляд. Например, какой результат нужно возвратить, если прямые параллельны? Как поступить, если прямые совпадают и «точкой пересечения» является вся прямая? Если одна из прямых расположена горизонтально, в процессе решения уравнений придется выполнить деление на нуль. Если прямые *почти* параллельны, точка пересечения находится так далеко от начала координат, что при вычислении ее местоположения возникнет арифметическое переполнение. Ситуация значительно усложняется при поиске пересечения двух отрезков, поскольку тогда возникает много дополнительных случаев, которые нужно отслеживать.

Если вы новичок в области реализации алгоритмов вычислительной геометрии, я рекомендую вам изучить книгу [O'R01], в которой содержатся практические советы и законченные реализации основных алгоритмов вычислительной геометрии и структур данных.

Мы имеем дело с двумя разными понятиями: геометрической вырожденностью и численной устойчивостью. Под *вырожденностью* (degeneracy) понимаются частные случаи, требующие специальных подходов, — например, такие, в которых две прямые имеют несколько точек пересечения. Существуют три подхода к решению проблемы вырожденности:

- *игнорировать ее.*

  Выдвигаем рабочую гипотезу, что наша программа будет работать правильно только в тех случаях, когда никакие три точки не коллинеарны, никакие три прямые не пересекаются в одной точке, пересечения отрезков не происходит в их конечных точках и т. д. Это, пожалуй, самый распространенный подход, который я могу рекомендовать для краткосрочных проектов, если допустимы частые отказы программы. Недостаток такого подхода обусловлен тем фактом, что самые интересные данные часто поступают из точек, выбранных на сетке, которая имеет тенденцию быть сильно вырожденной;

- *подделать невырожденность.*

  Случайным образом внесите беспорядок в данные, чтобы сделать их невырожденными. Перемещая каждую точку на небольшое расстояние в случайном направле-

нии, можно исключить вырожденность данных, не создавая при этом слишком много новых проблем. Это первое, что вы должны попробовать, когда определите, что частота отказов вашей программы превышает допустимый уровень. Недостаток такого подхода состоит в том, что случайные изменения могут исказить входные данные неприемлемым для вашего приложения образом. Существуют способы внесения «символических» возмущений в данные для непротиворечивого исключения вырожденности, но корректное применение этих способов требует серьезных исследований;

- *обрабатывать вырожденность.*

  Программу можно сделать более устойчивой, добавив в нее специальный код для обработки каждого частного случая. Такой подход может быть оправдан, если его осуществлять с начала разработки программы, но недопустимо добавлять «заплатки» при каждом сбое программы. Если вы решительно настроены на применение этого подхода, будьте готовы вложить в его реализацию значительные усилия.

В геометрических вычислениях часто применяются арифметические операции с плавающей точкой, из-за чего могут возникнуть такие проблемы, как переполнение и потеря точности. Существуют три основных подхода к обеспечению численной устойчивости:

- *использование целочисленных арифметических операций.*

  Принудительное размещение всех представляющих интерес точек на целочисленной сетке позволит выполнять точные проверки двух чисел на равенство или двух отрезков на пересечение. При этом нужно отдавать себе отчет в том, что точка пересечения двух прямых не всегда будет находиться строго на сетке. Тем не менее это самый простой и лучший способ при условии, что он дает приемлемые результаты;

- *использование действительных чисел двойной точности.*

  Используя числа двойной точности с плавающей точкой, можно уменьшить количество численных ошибок. Однако лучше всего для представления данных использовать действительные числа одинарной точности, а затем применять двойную точность для промежуточных вычислений;

- *использование арифметических операций произвольной точности.*

  Алгоритм, использующий такой подход, несомненно, даст правильные результаты, но будет работать очень медленно, поскольку операции с высокой точностью выполняются на несколько порядков медленнее, чем стандартные арифметические операции с плавающей точкой. При этом тщательный анализ может свести к минимуму необходимость в арифметических операциях с высокой точностью и, следовательно, потерю производительности.

Наилучший подход к разработке устойчивого программного обеспечения для решения задач вычислительной геометрии — это создание приложений на основе небольшого набора алгоритмов вычислительной геометрии, выполняющих большой объем вычислений низкого уровня. В число таких алгоритмов входят следующие:

- *вычисление площади треугольника.*

  Площадь $A(t)$ треугольника $t = (a, b, c)$ действительно равна половине произведения его основания на высоту, но вычисление длины основания и высоты требует кропотливых расчетов с применением тригонометрических функций. Намного лучше вычислить *двойную* площадь, используя следующую формулу определителя (определители обсуждаются в *разд. 16.4*):

$$2 \cdot A(t) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y.$$

  Эта формула обобщает задачу вычисления $d!$-кратного объема многогранника в $d$-мерном пространстве. Таким образом, в трехмерном пространстве шестикратный ($3! = 6$) объем четырехгранника $t = (a, b, c, d)$ будет равен

$$6 \cdot A(t) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}.$$

  Поскольку эти значения могут быть отрицательными, в дальнейших вычислениях следует использовать их модули.

  Концептуально самым простым способом вычисления площади многоугольника (или многогранника) будет выполнение его триангуляции с последующим суммированием площадей всех полученных треугольников. Реализацию изящного алгоритма, в котором не используется триангуляция, вы найдете в [O'R01] и [SR03];

- *выяснение местоположения точки.*

  Как расположена точка $c$ относительно прямой $l$? Чтобы дать корректный ответ на этот вопрос, следует представить прямую $l$ в виде направленной линии, проходящей через точку $a$ раньше, чем через точку $b$, и выяснить, слева или справа от этой линии находится точка $c$.

  Решить эту задачу можно, используя знак площади треугольника, вычисленной приведенным только что способом. Если площадь треугольника $t(a, b, c) > 0$, значит, точка $c$ находится слева от $\overline{ab}$. Если площадь треугольника $t(a, b, c) = 0$, значит, точка $c$ расположена на $\overline{ab}$. Наконец, если площадь треугольника $t(a, b, c) < 0$, значит, точка $c$ находится справа от $\overline{ab}$. Этот подход легко обобщается для трех измерений, где знак площади, показывает местоположение точки $d$ над ориентированной плоскостью $(a, b, c)$, под плоскостью или на ней;

- *проверка пересечения прямой и отрезка.*

  Приведенный способ выяснения расположения точки также можно использовать и для проверки пересечения прямой с отрезком. Такое пересечение имеет место тогда и только тогда, когда одна конечная точка отрезка находится слева от прямой, а вторая — справа. Проверка пересечения отрезков является сходной задачей. Во-

прос, пересекаются ли два отрезка, если они имеют общую концевую точку, представляет типичную ситуацию, в которой возникают проблемы с вырожденностью;

- *выяснение, находится ли точка внутри круга.*

  Необходимость в проверке, находится ли точка $d$ внутри круга, определяемого точками $a$, $b$ и $c$, возникает во всех алгоритмах триангуляции Делоне. Соответствующий алгоритм может быть использован в качестве надежного способа сравнения расстояний. Предполагая, что обозначение точек $a$, $b$, и $c$ идет против часовой стрелки, вычислим следующий определитель:

$$incircle(a,b,c) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix}.$$

  Значение этого определителя будет нулевым, если все четыре точки лежат на круге, положительным, если точка $d$ лежит внутри круга, и отрицательным, если точка $d$ находится вне круга.

Прежде чем пытаться создать свою собственную реализацию, ознакомьтесь с уже существующими, включая указанные в *подразделе «Реализации»*.

### Реализации

Библиотеки CGAL (**www.cgal.org**) и LEDA (см. *разд. 22.1.1*) содержат реализации алгоритмов вычислительной геометрии, написанные на языке C++. С библиотекой LEDA легче работать, но библиотека CGAL полнее и к тому же бесплатна. Если вы разрабатываете серьезное геометрическое приложение, вам следует ознакомиться с возможностями этих библиотек, прежде чем создавать свою собственную реализацию.

В книге [O'R01] представлены реализации на языке C большинства алгоритмов, обсуждаемых в этом разделе. Подробности см. на веб-сайте **http://cs.smith.edu/~jorourke/books/CompGeom/CompGeom.html**. Хотя эти реализации были созданы в основном для ознакомительных целей, а не для коммерческого применения, они надежны и годятся для небольших приложений.

Библиотека Core (**http://cs.nyu.edu/exact/**) содержит API-интерфейс, в котором используется подход EGC (Exact Geometric Computation — точные геометрические вычисления) к созданию численно устойчивых алгоритмов. Эту библиотеку можно использовать с небольшими изменениями в любой программе на языке C/C++, с легкостью поддерживая три уровня точности: машинную точность, произвольную точность и гарантированную точность.

Устойчивую реализацию основных алгоритмов вычислительной геометрии на языке C++, представленную в работе [She97], можно загрузить по адресу **http://www.cs.cmu.edu/~quake/robust.html**.

#### Примечания

Книга [O'R01] — прекрасное практическое введение в вычислительную геометрию. В ней особое внимание уделяется тщательной и правильной реализации алгоритмов. Библиотека

LEDA (см. [MN99]) является еще одним отличным образцом для подражания для тех, кто собирается разрабатывать собственные реализации.

В работе [SY18] и черновике книги на ее основе (см. [MY07]) дается замечательный обзор методов построения устойчивых реализаций. В работе [KMP+04] дано графическое представление проблем, которые могут возникать при использовании арифметических операций над действительными числами в алгоритмах вычислительной геометрии для поиска выпуклой оболочки. Управляемое внесение возмущений в данные (см. [MOS11]) является более новым подходом к обеспечению устойчивых вычислений. В работах [She97] и [FvW93] представлены результаты подробных исследований стоимости арифметических операций произвольной точности для геометрических вычислений. Если использовать эти операции с должной осторожностью, можно обеспечить приемлемую эффективность вычислений при их полной устойчивости.

### Родственные задачи

Выявление пересечений (см. *разд. 20.8*), конфигурации прямых (см. *разд. 20.15*).

## 20.2. Выпуклая оболочка

**Вход.** Множество $S$ из $n$ точек в $d$-мерном пространстве.

**Задача.** Найти наименьший выпуклый многоугольник (или многогранник), содержащий все точки множества $S$ (рис. 20.2).

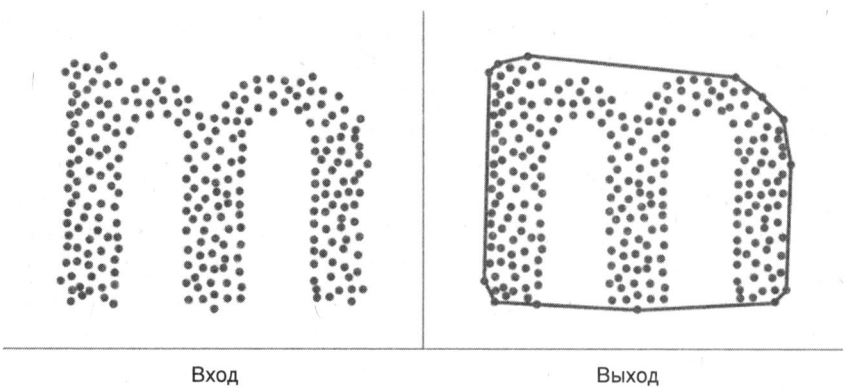



**Рис. 20.2.** Выпуклая оболочка

### Обсуждение

Задача построения выпуклой оболочки является самой важной элементарной задачей вычислительной геометрии — точно так же, как задача сортировки считается самой важной задачей комбинаторных алгоритмов. Важность этой задачи обусловлена тем, что создание выпуклой оболочки позволяет получить приблизительное представление о форме или размере набора данных.

Построение выпуклой оболочки является шагом предварительной обработки во многих алгоритмах вычислительной геометрии. Рассмотрим, например, задачу поиска диаметра набора точек — другими словами, поиска пары точек с наибольшим расстоянием

между ними. Диаметр равен расстоянию между двумя точками, расположенными на выпуклой оболочке. Алгоритм вычисления диаметра (имеющий время исполнения $O(n \lg n)$) сначала создает выпуклую оболочку, после чего для каждой вершины оболочки находит на оболочке вершину, наиболее от нее удаленную. Для быстрого перемещения от одного конца диаметра оболочки к следующему можно использовать «метод штангенциркуля», всегда двигаясь вдоль оболочки по часовой стрелке.

Существует почти такое же количество алгоритмов для построения выпуклой оболочки, как и алгоритмов сортировки. При выборе наиболее подходящего из них ответьте на следующие вопросы.

- *С каким количеством измерений вы имеете дело?*

  С выпуклыми оболочками довольно легко работать в двух- и даже трехмерном пространстве. Но некоторые предположения, справедливые для небольшого количества измерений, становятся недействительными при увеличении их количества. Например, в двумерном пространстве любой многоугольник с $n$ вершинами имеет ровно $n$ ребер. Но с ростом количества измерений всего лишь до трех, зависимость между количеством вершин и граней становятся более сложной. Например, куб имеет восемь вершин и шесть граней, а восьмигранник — шесть вершин и восемь граней. Поэтому при выборе структур данных, представляющих оболочки, принципиально важно, требуется ли нам всего лишь найти точки оболочки или мы должны построить многогранник. Следует иметь в виду такие особенности многомерных пространств, если при решении своей задачи вы будете вынуждены работать с ними.

  Итак, существуют простые алгоритмы построения выпуклой оболочки с временем исполнения $O(n \log n)$ для двумерных и трехмерных случаев, но с увеличением размерности задача становится более сложной. При создании выпуклых оболочек в многомерных пространствах используется базовый подход, называемый *заворачиванием подарка* (gift-wrapping). Обратите внимание, что трехмерный выпуклый многогранник состоит из двумерных *граней*, соединенных одномерными *ребрами*. Каждое ребро соединяет ровно две грани. Заворачивание подарка начинается с определения начальной грани, связанной с самой нижней вершиной, после чего от этой грани выполняется поиск в ширину, чтобы найти следующие грани. Каждое ребро $e$, принадлежащее грани, должно быть общим с какой-то другой гранью. Перебрав все $n$ точек, мы можем выяснить, какая точка определяет другую грань ребра $e$. Образно говоря, мы «оборачиваем» точки по одной грани за раз — наподобие сгибания оберточной бумаги по ребрам — пока она не закроет первую точку.

  Для обеспечения эффективности алгоритма необходимо, чтобы каждое ребро исследовалось только один раз. Время исполнения корректно реализованного алгоритма для $d$ измерений равно $O(n\Phi_{d-1} + \Phi_{d-2} \lg \Phi_{d-2})$, где $\Phi_{d-1}$ — количество граней, а $\Phi_{d-2}$ — количество ребер в выпуклой оболочке. Но если выпуклая оболочка очень сложна, то время исполнения может ухудшиться до $O\left(n^{\lfloor d/2 \rfloor + 1}\right)$. Поэтому используйте готовые реализации, а не пытайтесь разработать свою собственную.

- *Данные представлены в виде вершин или в виде полупространств?*

  Задача поиска пересечения набора $n$ полупространств в $d$-мерном пространстве (каждое из которых содержит начало координат) аналогична задаче поиска выпуклых

оболочек из $n$ точек в $d$-мерном пространстве. Таким образом, для решения обеих задач достаточно одного алгоритма. Требуемое преобразование рассматривается в *разд. 20.15*. Когда внутренняя точка не задана, задача поиска пересечения полуплоскостей перестает быть аналогичной задаче поиска выпуклой оболочки, поскольку при пустом пересечении полуплоскостей возникают невозможные экземпляры.

- *Сколько точек может содержать оболочка?*

  Если набор точек сгенерирован псевдослучайным образом, большинство точек будет, скорее всего, находиться внутри оболочки. Практическую эффективность программ построения выпуклых оболочек на плоскости можно повысить, используя то обстоятельство, что самая левая, самая правая, самая верхняя и самая нижняя точки должны находиться на выпуклой оболочке. При этом мы имеем набор из трех или четырех разных точек оболочки, определяющих треугольник или четырехугольник. Любая точка внутри этой области *не может* находиться на выпуклой оболочке и, следовательно, ее можно отбросить путем линейного перебора точек. В идеале после этого останется небольшое количество точек, которые можно просмотреть с помощью алгоритма построения выпуклой оболочки.

  Этот прием можно применять в многомерных пространствах, но с увеличением количества измерений его эффективность снижается.

- *Как выяснить форму набора точек?*

  Хотя выпуклая оболочка дает приблизительное представление о форме, многие подробности теряются. Например, выпуклая оболочка для набора точек в форме буквы $m$ (см. рис. 20.2) неотличима от выпуклой оболочки для буквы $w$. Более общими структурами, которые можно параметризировать, чтобы сохранить «невыпуклость» множества точек, являются *альфа-очертания* (alpha shapes). Ссылки на справочный материал и реализации соответствующих алгоритмов приводятся далее.

Основным алгоритмом построения выпуклой оболочки на плоскости является *сканирование по Грэхему*. Алгоритм Грэхема начинает работу с точки $p$, заведомо находящейся на выпуклой оболочке (например, с точки, имеющей самое меньшее значение координаты $x$), а затем сортирует остальные точки в порядке возрастания полярного угла, измеряемого против часовой стрелки относительно этой точки. Начиная с частичной выпуклой оболочки, состоящей из точки $p$ и точки, имеющей наименьший полярный угол, алгоритм выполняет перебор всех точек против часовой стрелки с добавлением в оболочку новых точек. При этом если угол между очередной точкой и последним ребром оболочки меньше чем $180^\circ$, то эта точка вставляется в оболочку. А если этот угол больше $180^\circ$, то цепочка вершин, начинающаяся с последнего ребра оболочки, подлежит удалению для сохранения выпуклости. Общее время исполнения алгоритма равно $O(n \lg n)$, поскольку сортировка точек вокруг $p$ является узким местом.

Эту процедуру алгоритма Грэхема можно также использовать для создания *простого* (не имеющего самопересечений) многоугольника, проходящего через все заданные точки. Для этого сортируем точки вокруг вершины $v$, но вместо проверки углов соединяем точки в том порядке, в каком они отсортированы. В результате получим многоугольник без самопересечений, но нередко с большим количеством некрасивых выступающих частей.

Алгоритм заворачивания подарка значительно упрощается в двумерном пространстве, когда каждая «грань» становится ребром, а каждое «ребро» — вершиной многоугольника и «поиск в ширину» выполняется вдоль оболочки: по или против часовой стрелки. Время исполнения алгоритма заворачивания подарка для двух измерений равно $O(nh)$, где $h$ — количество вершин в выпуклой оболочке. Я рекомендую использовать алгоритм Грэхема за исключением случаев, когда вы знаете заранее, что оболочка содержит лишь небольшое количество вершин.

### Реализации

Библиотека CGAL (**www.cgal.org**) включает в себя реализации на языке C++ разнообразных алгоритмов построения выпуклой оболочки для произвольного количества измерений. Альтернативные реализации на языке C++ алгоритмов построения выпуклых оболочек содержатся в библиотеке LEDA (см. *разд. 22.1.1*).

Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см.[BDH97]), оптимизированная для работы в диапазоне от двух до восьми измерений. Программа написана на языке C и может также создавать триангуляции Делоне, диаграммы Вороного и пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**.

В книге [O'R01] приведена устойчивая реализация алгоритма Грэхема для двух измерений и реализация с временем исполнения $O(n^2)$ инкрементального алгоритма построения выпуклых оболочек в трех измерениях. Программы этой реализации написаны на языках C и Java. Подробности см. в *разд. 22.1.9*.

Реализацию Hull для построения выпуклых оболочек в многомерных пространствах также можно использовать для построения альфа-очертаний. Программа доступна на веб-сайте **http://www.netlib.org/voronoi/hull.html**.

Для перечисления вершин пересекающихся полупространств с большим количеством измерений требуются другие реализации. Программа lrs (**http://cgm.cs.mcgill.ca/~avis/C/lrs.html**) представляет собой арифметически устойчивую реализацию на языке ANSI C алгоритма поиска в обратном направлении Ависа — Фукуды для решения задач перечисления вершин и построения выпуклых оболочек. Так как обход многогранников выполняется неявным образом, и они явно не сохраняются в памяти, то иногда с помощью этой программы можно решить даже задачи с очень большим размером вывода.

#### *Примечания*

Роль плоских выпуклых оболочек в вычислительной геометрии тождественна роли сортировки в комбинаторике. Подобно сортировке, задача построения выпуклой оболочки является фундаментальной, и разнообразные подходы ведут к созданию интересных или оптимальных алгоритмов. Так, алгоритмы Quickhull и mergehull представляют собой примеры алгоритмов построения выпуклой оболочки, на создание которых разработчиков вдохновили алгоритмы сортировки (см. [PS85]). Простая конструкция из точек на параболе, рассмотренная в *разд. 11.2.4*, сводит задачу построения выпуклой оболочки к задаче сортировки, так что теоретическая нижняя граница сортировки позволяет сделать вывод, что для построения плоской выпуклой оболочки требуется время $\Omega(n \lg n)$. Более сильная нижняя граница установлена в работе [Yao81].

Подробное обсуждение алгоритмов Грэхема (см. [Gra72]) и Джарвиса (см. [Jar73]) содержится в книгах [dBvKOS08], [CLRS09], [O'R01] и [PS85]. Алгоритм для построения плоской выпуклой оболочки (см. [KS86]) выдает оптимальное решение за время $O(n \lg h)$, где $h$ — количество вершин оболочки, и обеспечивает наилучшую производительность как алгоритма Грэхема, так и алгоритма заворачивания подарка. Вычисление планарных выпуклых оболочек можно эффективно выполнять «на месте» — т. е. без выделения дополнительной памяти (см. [BIK+04]). В работе [Sei18] представлен замечательный обзор алгоритмов построения выпуклой оболочки и их вариантов, в особенности для многомерных пространств.

*Топология* — это наука, изучающая геометрические формы. Введение в вычислительную топологию приводится в книге [EH10]. Альфа-очертания, обсуждаемые в работе [EKS83], дают хорошее представление о форме набора точек. Обобщение оболочек этого типа для трехмерного пространства и соответствующая реализация приведены в работе [EM94].

Алгоритмы поиска в обратном направлении для построения выпуклых оболочек эффективно работают и в многомерных пространствах (см. [AF96]). При удачном построении отображения в пространство с большим количеством измерений (см. [ES86]) задачу создания диаграммы Вороного в $d$-мерном пространстве можно свести к задаче создания выпуклой оболочки в $(d + 1)$-мерном пространстве. Подробности см. в *разд. 20.4*.

Алгоритмы, предназначенные для работы с выпуклыми оболочками, используют специальные структуры данных, которые позволяют выполнять вставку и удаление произвольных точек и при этом непрерывно отображают текущую выпуклую оболочку. Стоимость этих операций была понижена до логарифмического амортизированного времени (см. [JB19]).

### Родственные задачи

Сортировка (см. *разд. 17.1*), диаграммы Вороного (см. *разд. 20.4*).

## 20.3. Триангуляция

**Вход.** Набор точек, многоугольник или многогранник.

**Задача.** Разбить внутреннюю часть набора точек или многогранника на треугольники (рис. 20.3).

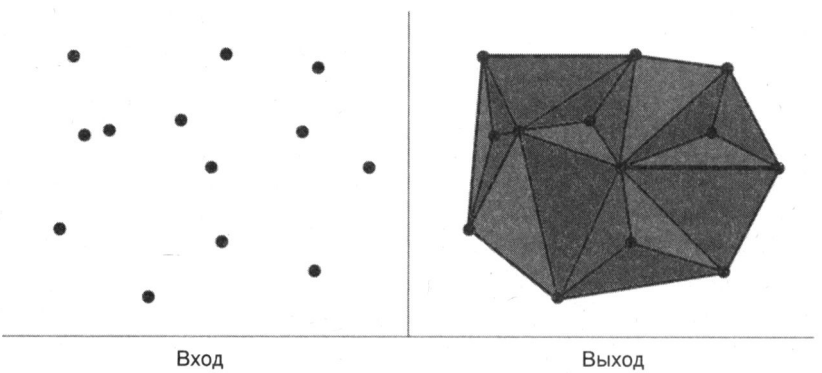



**Рис. 20.3.** Триангуляция

### Обсуждение

Хорошим первым шагом в обработке сложных геометрических объектов стало бы разбиение их на простые геометрические фигуры. Это обстоятельство делает триангуляцию фундаментальной задачей вычислительной геометрии, поскольку самым простым двумерным геометрическим объектом является треугольник. В число классических применений триангуляции входят анализ методом конечных элементов и компьютерная графика.

Одним интересным применением триангуляции является интерполирование на поверхности. Допустим, что имеются значения высоты горы в множестве точек $(x, y, z)$. Как установить приблизительную высоту горы $z'$ в заданной точке $q = (x', y')$? Мы можем проецировать точки, в которых замерена высота, на плоскость, а потом выполнить триангуляцию. Плоскость будет разбита на треугольники, позволяющие приблизительно вычислить высоту точки $q$ путем интерполирования высот (координат $z$) вершин треугольника, содержащего эту точку. Кроме того, триангуляция и соответствующие значения высот определяют поверхность горы, пригодную для создания компьютерного изображения.

Триангуляция на плоскости выполняется путем соединения точек непересекающимися отрезками до тех пор, пока добавление новых отрезков не становится невозможным. При этом вы должны ответить на следующие вопросы.

- *Триангулируется набор точек или многоугольник?*

  Обычно приходится триангулировать набор точек, как в рассмотренной ранее задаче интерполирования поверхности. Для решения этой задачи нужно сначала создать выпуклую оболочку для имеющегося набора точек, а потом разбить внутренюю область полученной оболочки на треугольники.

  Самый простой алгоритм обработки набора точек, имеющий время исполнения $O(n \lg n)$, сначала сортирует точки по $x$-координате. Потом отсортированные точки выбираются слева направо согласно алгоритму построения выпуклой оболочки, описанному в *разд. 4.1*. При этом выполняется триангуляция путем добавления ребра к каждой новой отрезанной от оболочки точке.

- *Имеет ли значение внешний вид треугольников триангуляции?*

  Обычно вход можно разбить на треугольники многими разными способами. Рассмотрим набор из $n$ точек в выпуклой оболочке на плоскости. Самый простой способ выполнить триангуляцию этих точек — добавить расположенные веером диагонали, идущие от первой точки до всех остальных n – 1 точек. Но в таком случае обычно появляются «узкие» треугольники.

  Во многих приложениях необходимо избегать «узких» треугольников или, что то же самое, минимизировать количество очень острых углов триангуляции. Триангуляция набора точек *методом Делоне* максимизирует минимальный угол по всем возможным вариантам триангуляции. Правда, это не совсем то, что нам требуется, но, в принципе, весьма близко. Вообще, триангуляция методом Делоне имеет достаточно много других интересных свойств, чтобы я рекомендовал предпочесть ее другим методам триангуляции. Кроме того, используя описанные далее реализации, такую триангуляцию можно выполнить за время $O(n \lg n)$.

- *Как можно улучшить результат триангуляции?*

  Каждое внутреннее ребро любой триангуляции принадлежит двум треугольникам. Четыре вершины, определяющие эти два треугольника, образуют выпуклый или невыпуклый четырехугольник. Достоинство выпуклого варианта заключается в том, что при замене внутреннего ребра на ребро, связывающее две другие вершины, получается другая триангуляция.

  Иначе говоря, мы имеем локальную операцию замены ребер, посредством которой можно модифицировать и, не исключено, улучшать полученную триангуляцию. В самом деле, триангуляцию Делоне можно получить из любой триангуляции, удаляя «узкие» треугольники до тех пор, пока не будут исчерпаны все возможности для локальных улучшений.

- *Какова размерность задачи?*

  Трехмерные задачи обычно гораздо сложнее, чем двумерные. Обобщение процедуры триангуляции до трех измерений заключается в разбиении пространства на четырехвершинные четырехгранники путем добавления непересекающихся граней. Трудность заключается в том, что некоторые многогранники нельзя разбить на четырехгранники, не добавляя дополнительные вершины. Кроме этого, задача выяснения, существует ли такое разбиение на четырехгранники, является NP-полной, поэтому вы можете смело добавлять новые вершины, чтобы упростить задачу.

- *Каковы ограничивающие условия на входные данные?*

  При триангуляции многоугольника или многогранника можно добавлять только ребра, не пересекающие какие-либо внешние грани. Вообще говоря, может существовать набор дополнительных ограничений или препятствий, которые нельзя пересекать добавляемыми ребрами. Самой лучшей триангуляцией при таких условиях будет, скорее всего, *триангуляция Делоне с ограничениями.*

- *Можно ли добавлять дополнительные точки или перемещать существующие?*

  Когда форма треугольников действительно имеет значение, может оказаться целесообразным добавить небольшое количество дополнительных точек Штейнера к набору данных, чтобы облегчить создание триангуляции, удовлетворяющей заданному условию, — например, требованию отсутствия очень острых углов. Как упоминалось ранее, для триангуляции некоторых многогранников точки Штейнера добавлять необходимо.

Чтобы выполнить триангуляцию выпуклого многоугольника за линейное время, просто выберем случайную начальную вершину $v$ и соединим ее ребрами со всеми остальными вершинами многоугольника. Так как многоугольник выпуклый, то мы можем быть уверены в том, что добавляемые ребра не будут пересекать грани многоугольника. Самый простой алгоритм для выполнения общей триангуляции многоугольника проверяет каждое из $O(n^2)$ возможных ребер на пересечение с граничными или ранее вставленными ребрами и вставляет его только в случае отсутствия такого пересечения. Существуют пригодные для практического применения алгоритмы с временем исполнения $O(n \lg n)$, а также представляющие теоретический интерес алгоритмы с линейным временем исполнения. Подробности см. в *подразделах «Реализации»* и *«Примечания»*.

### Реализации

Пакет Triangle, разработанный Джонатаном Шевчуком (Jonathan Shewchuk), содержит набор процедур на языке C для генерирования триангуляций Делоне, триангуляций Делоне с ограничениями (т. е. триангуляций, в которых некоторые ребра вставляются в принудительном порядке), а также триангуляций Делоне, обеспечивающих качественный результат (т. е. таких, в которых очень острые углы исключаются с помощью вставки промежуточных точек). Этот высокопроизводительный и устойчивый пакет широко используется для анализа методом конечных элементов. Если бы мне требовалась реализация алгоритма двумерной триангуляции, то я начал бы с программ из пакета Triangle. Загрузить пакет можно с веб-страницы **http://www.cs.cmu.edu/~quake/triangle.html**.

Программа Sweep2, написанная на языке C, прекрасно подходит для создания двумерных диаграмм Вороного и триангуляций Делоне. С ней особенно удобно работать, когда вам требуется лишь триангуляция Делоне точек на плоскости. Программа основана на *алгоритме заметающей прямой* (sweepline alogirthm) для создания диаграмм Вороного (см. [For87]) и доступна в онлайновом хранилище Netlib (см. *разд. 2.1.4*) по адресу **https://www.netlib.org/voronoi/**.

Похоже, что предпочитаемой многими исследователями программой для разложения трехмерных многогранников на четырехгранники (см. [Si15]) является программа TetGen (**http://wias-berlin.de/software/tetgen/**). Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *разд. 22.1.1*) содержат реализации на языке C++ самых разных алгоритмов триангуляции в двух- и трехмерных пространствах, включая триангуляции Делоне как с ограничениями, так и с самым дальним расположением.

В многомерных пространствах триангуляции Делоне представляют собой частный случай выпуклых оболочек. Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см. [BDH97]), оптимизированная для работы в диапазоне от двух до восьми измерений. Программа написана на языке C и может также создавать триангуляции Делоне, диаграммы Вороного и пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**. В качестве альтернативы доступна программа Hull, предназначенная для построения выпуклых оболочек в многомерных пространствах (**https://www.netlib.org/voronoi/hull.html**).

#### Примечания

Шазель (Chazelle) предоставил линейный алгоритм триангуляции простого многоугольника (см. [Cha91]), что является важным теоретическим результатом, поскольку триангуляция была узким местом для многих других геометрических алгоритмов. Но реализация этого алгоритма является исключительно трудной задачей, так что он больше подходит в качестве доказательства существования триангуляционного разбиения. Но известно о существовании более простого рандомизированного алгоритма (см. [AGR01]). Первый алгоритм триангуляции многоугольников с временем исполнения $O(n \lg n)$ был представлен в работе [GJPT78]. После создания этого алгоритма (но раньше появления алгоритма Шазеля) Тарьян и ван Вик разработали еще один алгоритм с временем исполнения $O(n \lg \lg n)$ (см. [TW88]). Обзор результатов в области триангуляции наборов точек и многоугольников представлен в [BSA18].

В число книг по предметам триангуляции Делоне и генерирования качественных сеток входят [AKL13] и [SDC16]. Для исследователей, интересующихся генерированием сеток и решеток, проводится ежегодная конференция International Meshing Roundtable. Отличными обзорами методов генерирования сеток можно признать работы [Ber02] и [Ede06].

Алгоритмы с линейным временем исполнения для триангуляции монотонных многоугольников давно известны (см. [GJPT78]) и составляют базу для алгоритмов триангуляции простых многоугольников. *Монотонным* называется многоугольник, для которого существует такое направление, что любая прямая, идущая в этом направлении, пересекает многоугольник самое большее в двух точках.

Хорошо изученный класс оптимальных триангуляций включает в себя решения, стремящиеся минимизировать суммарную длину использованных ребер. В работе [MR06] доказано, что такая задача является NP-полной. После этого внимание исследователей переключилось на доказуемо хорошие аппроксимирующие алгоритмы (см. [RW18]). Триангуляцию с минимальным весом выпуклого многоугольника можно выполнить за время $O(n^3)$, используя динамическое программирование.

### Родственные задачи

Диаграммы Вороного (см. *разд. 20.4*), разбиение многоугольников (см. *разд. 20.11*).

## 20.4. Диаграммы Вороного

**Вход.** Множество $S$ точек $p_1, ..., p_n$.

**Задача.** Разбить пространство на области таким образом, чтобы все точки в области вокруг точки $p_i$ были ближе к этой точке, чем к любой другой точке множества $S$ (рис. 20.4).

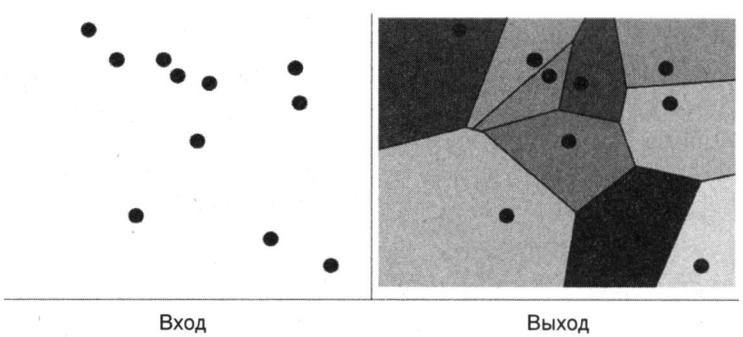



**Рис. 20.4.** Диаграмма Вороного

### Обсуждение

Диаграмма Вороного представляет области влияния вокруг точек из имеющегося набора $S$. Если эти точки соответствуют, например, расположению ресторанов быстрого питания, то диаграмма Вороного $V(S)$ разбивает пространство на зоны, окружающие каждый ресторан. Для человека, живущего в конкретной зоне, соответствующий ресторан окажется ближайшим местом, где можно заказать горячий бутерброд.

Диаграммы Вороного имеют множество различных применений:

- *поиск ближайшей точки.*

  Поиск ближайшего соседа точки $q$ из фиксированного множества $S$ точек сводится к задаче поиска ячейки в $V(S)$, которая содержит такую точку (подробнее об этом рассказано в *разд. 20.5*);

- *размещение точек обслуживания.*

  Предположим, что владелец сети ресторанов быстрого питания хочет открыть еще один ресторан. Чтобы свести к минимуму конкуренцию с уже имеющимися в этой местности ресторанами, следует разместить новый ресторан как можно дальше от ближайшего ресторана. Искомое место всегда будет находиться в вершине диаграммы Вороного, и поэтому его можно найти за линейное время, выполнив поиск по таким вершинам;

- *наибольший пустой круг.*

  Допустим, что требуется найти большой неосвоенный земельный участок, чтобы построить фабрику. Условие, определяющее выбор места для ресторана быстрого питания, применимо и при поиске месторасположения нежелательных объектов. Иначе говоря, они должны располагаться как можно дальше от любого интересующего нас места. Вершина Вороного определяет центр наибольшего пустого круга вокруг точек;

- *разработка маршрута.*

  Если объекты множества $S$ являются препятствиями, которых следует избегать, то ребра диаграммы Вороного $V(S)$ определяют возможные маршруты с максимальным расстоянием до препятствий. Таким образом, самый «безопасный» путь среди препятствий будет проходить по ребрам диаграммы Вороного;

- *улучшение триангуляции.*

  Выполняя триангуляцию набора точек, мы обычно хотим получить треугольники без очень острых углов и стараемся избегать «узких» треугольников. Триангуляция Делоне максимизирует минимальный угол по всем возможным триангуляциям, и ее можно сформулировать как задачу, близкую к диаграмме Вороного (подробности вы найдете в *разд. 20.3*).

Каждое ребро диаграммы Вороного является отрезком серединного перпендикуляра линейного сегмента между двумя точками в множестве $S$, поскольку эта линия делит плоскость посередине между двумя точками. Самый простой метод создания диаграмм Вороного — это рандомизированное инкрементальное построение. Чтобы добавить новую точку $p$ в диаграмму, мы находим содержащую ее ячейку и добавляем серединные перпендикуляры, которые отделяют эту точку $p$ от других точек, определяющих области, подвергающиеся влиянию. Если точки вставляются в произвольном порядке, то каждая такая вставка повлияет, скорее всего, лишь на небольшое количество областей.

Однако самым лучшим методом можно признать алгоритм Форчуна, основанный на *методе заметающей прямой*. Его главное достоинство состоит в том, что для него существуют устойчивые реализации. Алгоритм отображает набор точек на плоскости

на набор конусов в трехмерном пространстве таким образом, что диаграмма Вороного определяется отображением конусов обратно на плоскость. Преимущества алгоритма Форчуна: оптимальное время исполнения $\Theta(n \log n)$, легкость реализации и отсутствие необходимости сохранять всю диаграмму в процессе заметания.

Между выпуклыми оболочками в $(d+1)$-мерном пространстве и триангуляциями Делоне (или, что эквивалентно, диаграммами Вороного) в $d$-мерном пространстве существует интересная связь. Отобразив каждую точку из $E^d$ на $E^d+1$:

$$(x_1, x_2, \ldots, x_d) \rightarrow \left( x_1, x_2, \ldots, x_d, \sum_{i=1}^{d} x_i^2 \right),$$

затем создав выпуклую оболочку этого $(d+1)$-мерного набора точек и, наконец, отобразив его обратно в $d$-мерное пространство, мы получим триангуляцию Делоне. Дополнительная информация содержится в *подразделе «Примечания»*, однако представленный здесь способ создания диаграмм Вороного в многомерных пространствах является самым лучшим. Программы построения выпуклых оболочек в многомерном пространстве рассматриваются в *разд. 20.2*.

На практике возникает несколько важных вариантов стандартной диаграммы Вороного:

- *неевклидовы метрики расстояний.*

  Вспомним, что диаграммы Вороного разбивают пространство на зоны влияния вокруг каждой заданной точки. До сих пор мы предполагали, что влияние измеряется евклидовым расстоянием, но это не всегда так. Например, время, чтобы добраться до ресторана на машине, зависит не от расстояния, а от того, как проложены дороги. Существуют эффективные алгоритмы для создания диаграмм Вороного при разных метриках;

- *диаграммы мощности.*

  Эти структуры разбивают пространство на зоны влияния вокруг точек, которые могут иметь разную мощность. Рассмотрим, например, сеть радиостанций, работающих на одной частоте. Зона влияния каждой станции зависит как от мощности ее передатчика, так и от расположения и мощности соседних станций;

- *диаграммы k-го порядка и диаграммы для самых дальних точек.*

  Идею разбиения пространства на зоны с каким-либо общим свойством можно распространить за пределы диаграмм Вороного для ближайших точек. Все точки в одной ячейке диаграммы Вороного $k$-го порядка имеют один и тот же набор ближайших соседних точек из множества $S$. В диаграммах для самых дальних точек все точки внутри определенной области имеют одну и ту же самую дальнюю точку из множества $S$. Расположение точек (см. *разд. 20.7*) на этих структурах позволяет быстро находить требуемые точки.

### Реализации

Для создания двумерных диаграмм Вороного и триангуляций Делоне широко используется написанная на языке C программа Sweep2. С ней очень просто работать, если вам требуется только диаграмма Вороного. Программа основана на алгоритме заме-

тающей прямой для создания диаграмм Вороного (см. [For87]). Загрузить ее можно с веб-страницы **https://www.netlib.org/voronoi**.

Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *разд. 22.1.1*) содержат реализации на языке C++ самых разных алгоритмов для создания диаграмм Вороного и триангуляций Делоне в двух- и трехмерных пространствах.

Диаграммы Вороного в пространствах с большим количеством измерений и диаграммы Вороного для самых дальних точек можно создавать как частный случай выпуклых оболочек в многомерном пространстве. Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см. [BDH97]), оптимизированная для работы в диапазоне от двух до восьми измерений. Программа написана на языке C и может создавать триангуляции Делоне, диаграммы Вороного, а также пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**. Альтернативным вариантом является программа Hull для создания выпуклых оболочек (**https://www.netlib.org/voronoi/hull.shar**).

#### *Примечания*

Свое название диаграммы Вороного получили по имени математика Г. Вороного, который сообщил о них в своем докладе в 1908 году. Однако Дирихле изучал этот вопрос в 1850 году, вследствие чего эти диаграммы иногда называют также разбиениями Дирихле.

Наиболее полное обсуждение диаграмм Вороного и их приложений содержится в книгах [OBSC00] и [AKL13]. Отличный обзор диаграмм Вороного и их вариантов — таких как диаграммы мощности — приводится в работе [For18]. Первый алгоритм создания диаграмм Вороного за время $O(n \lg n)$ основан на методе «разделяй и властвуй» (см. [SH75]). Подробное изложение алгоритма заметающей прямой Форчуна (см. [For87]), а также обсуждение связи между триангуляциями Делоне и $(d + 1)$-мерными выпуклыми оболочками (см. [ES86]) можно найти в [dBvKOS08] и [O'R01].

Для создания диаграммы Вороного $k$-го порядка пространство разбивается на области таким образом, что каждая точка такой области находится ближе всего к одному и тому же набору из $k$ точек. Представленный в работе [ES86] алгоритм позволяет создавать диаграммы Вороного $k$-го порядка за время $O(n^3)$. Выясняя местоположение точек в этой структуре, можно найти $k$ ближайших соседей заданной точки за время $O(k + \lg n)$. Обсуждение диаграмм Вороного $k$-го порядка содержится в книгах [O'R01] и [PS85].

### Родственные задачи

Поиск ближайшей точки (см. *разд. 20.5*), выяснение местоположения точки (см. *разд. 20.7*), поиск области (см. *разд. 20.6*).

## 20.5. Поиск ближайшей точки

**Вход.** Множество $S$ из $n$ точек в $d$-мерном пространстве, точка $q$.

**Задача.** Найти точку в $S$, ближайшую к точке $q$ (рис. 20.5).

### Обсуждение

Во многих геометрических приложениях возникает необходимость быстро найти точку, ближайшую к заданной точке. Классическим примером такой ситуации является

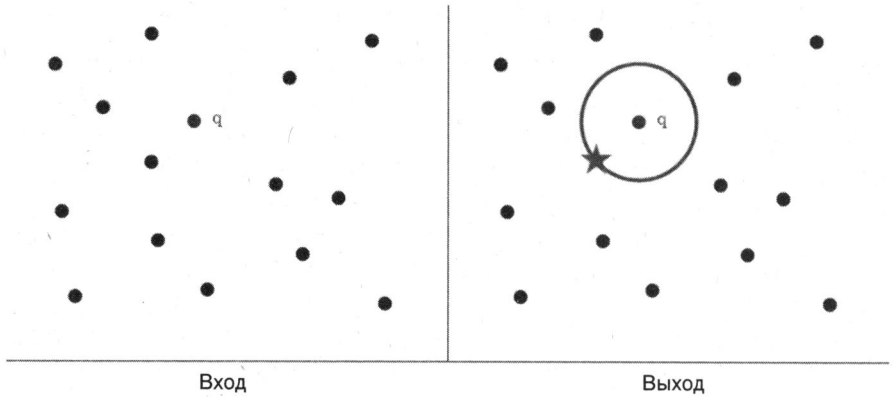



**Рис. 20.5.** Поиск ближайшей точки

разработка диспетчерской системы пожаротушения. Получив вызов, диспетчер должен найти ближайшую к пожару станцию, чтобы минимизировать время прибытия пожарной команды. Аналогичная ситуация возникает в любом приложении, где требуется сопоставление клиентов с поставщиками услуг.

Задача поиска ближайшего соседа также важна и в области классификации. Допустим, что у нас имеется база данных избирателей, в которой указаны их возраст, рост, вес, образование, уровень дохода и т. п. Про каждого избирателя также известно, какой из двух политических партий: демократической или республиканской — он симпатизирует. Нам требуется создать классификатор, позволяющий прогнозировать, как новый избиратель будет голосовать. Каждый избиратель в базе данных представлен точкой в $d$-мерном пространстве, помеченной названием партии. Простой классификатор избирателей можно создать, присваивая новой точке ту же метку, что и у ее ближайшего соседа.

Такие классификаторы, основанные на методе поиска ближайшего соседа, применяются достаточно широко. При сжатии изображений методом векторного квантования изображение разбивается на участки размером 8×8 пикселов. Метод использует заранее составленную библиотеку из нескольких тысяч элементов размером 8×8 пикселов и заменяет каждый участок изображения наиболее похожим элементом из библиотеки. Такой элемент представляется точкой в 64-мерном пространстве, ближайшей к точке, которая представляет рассматриваемый участок изображения. Сжатие, сопровождаемое небольшой потерей качества изображения, достигается заменой 64 пикселов на 12-битовый идентификатор самого похожего библиотечного элемента.

Решая задачу поиска ближайшего соседа, ответьте на следующие вопросы.

- *Каков размер пространства поиска?*

  Если набор данных состоит из небольшого количества точек (скажем, $n \leq 100$) или если планируется небольшое количество запросов, то самый простой подход и станет самым лучшим: сравниваем точку $q$ с каждой из $n$ точек набора данных. Более сложные методы стоит рассматривать только в том случае, когда требуется быстрое выполнение запросов для большого количества точек.

- *Какова размерность пространства?*

  По мере возрастания количества измерений поиск ближайшего соседа становится все труднее. Представленная в *разд. 15.6* структура данных kd-деревьев очень хорошо подходит для работы в пространствах с небольшим количеством измерений и даже с плоскостями. Тем не менее при количестве измерений, приближающемся к 20, поиск в kd-деревьях вырождается почти до линейного поиска. Поиск в многомерных пространствах становится трудным по той причине, что с увеличением размерности шар радиусом $r$, представляющий все точки на расстоянии $\leq r$ от центра, имеет все меньший объем по сравнению с объемом куба. Таким образом, любая структура данных, основанная на разбиении множества точек на подмножества внутри охватывающих сфер, будет становиться все менее эффективной по мере увеличения количества измерений. В больших количествах измерений лучше работают структуры данных на основе сфер, называемые *шаровыми деревьями*.

  Диаграммы Вороного в двумерном пространстве (см. *разд. 20.4*) предоставляют для поиска ближайшего соседа вполне эффективную структуру данных. Диаграмма Вороного на плоскости разбивает эту плоскость на ячейки таким образом, что каждая ячейка, содержащая точку $p$, содержит все множество точек, расположенных ближе к точке $p$, чем к любой другой точке множества $S$. Задача поиска ближайшего соседа точки $q$ сводится к задаче поиска ячейки диаграммы Вороного, содержащей точку $q$, и указания точки, связанной с этой ячейкой. Хотя диаграммы Вороного можно создавать в многомерных пространствах, с увеличением количества измерений их размер быстро возрастает до такой степени, что они становятся непригодными для использования.

- *Действительно ли нужен ближайший сосед?*

  Поиск *абсолютного* ближайшего соседа точки в многомерном пространстве представляет собой трудную задачу, для которой, скорее всего, нет решения лучше, чем линейный поиск (т. е. поиск методом исчерпывающего перебора). Но для решения этой задачи существуют эвристические алгоритмы, которые могут довольно быстро найти точку, расположенную достаточно близко от заданной точки.

  Один из подходов заключается в *понижении размерности* (dimension reduction). Существуют способы отображения любого набора из $n$ точек в $d$-мерном пространстве на пространство, имеющее $d' = O(\lg n/\varepsilon^2)$ измерений, таким образом, что расстояние до ближайшего соседа в пространстве с меньшим количеством измерений превышает расстояние до действительно ближайшего соседа примерно в $(1 + \varepsilon)$ раз. Отображение точек на $d'$ произвольных гиперплоскостей с большой вероятностью принесет желаемые результаты.

  Альтернативный подход заключается во внесении элемента случайности при поиске в структуре данных. Структура данных kd-деревьев позволяет выполнять эффективный поиск ячейки, содержащей точку $q$, — т. е. ячейки, граничные точки которой являются хорошими кандидатами на роль близких соседей. Теперь допустим, что мы ищем точку $q'$, которая является результатом небольшого случайного перемещения точки $q$. Мы попадем в другую ячейку поблизости, одна из граничных точек которой может оказаться еще более близким соседом точки $q$. Многократное повторе-

ние подобных рандомизированных запросов позволит нам эффективно потратить на улучшение результата столько времени, сколько мы можем себе позволить.

- *Набор данных является статическим или динамическим?*

  Будут ли в вашем приложении выполняться операции вставки или удаления точек данных? Если вставка и удаление выполняются нечасто, то после каждой такой операции имеет смысл заново строить структуру данных. В противном случае следует использовать версию kd-дерева, поддерживающую операции вставки и удаления.

Граф ближайшего соседа для множества $S$ из $n$ точек связывает каждую вершину со своим ближайшим соседом. Такой граф является подграфом триангуляции Делоне, и поэтому его можно вычислить за время $O(n \log n)$. Это довольно хороший результат, поскольку только поиск ближайшей пары точек множества $S$ занимает время $\Theta(n \log n)$.

Задача поиска пары ближайших точек в одномерном пространстве сводится к сортировке. В отсортированном наборе чисел пара ближайших точек соответствует двум числам, расположенным рядом друг с другом, поэтому нам нужно лишь найти минимальное расстояние между $n-1$ смежными точками. Предельный случай возникает, когда расстояние между ближайшими точками равно нулю, откуда следует, что элементы не являются уникальными.

### Реализации

Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Эти реализации дают хорошие результаты для поиска среди сотен тысяч точек в пространствах, имеющих до 20 измерений. Библиотека поддерживает все $l_p$-нормы, включая евклидово и манхэттенское расстояния. Загрузить библиотеку можно по адресу **http://www.cs.umd.edu/~mount/ANN/**. Если бы мне пришлось решать задачу поиска ближайшего соседа, я начал бы с этой библиотеки.

Компания Spotify разработала библиотеку Annoy на языке C++ с привязкой для языка Python (**https://github.com/spotify/annoy**). Библиотека основана на деревьях случайных проекций и предназначена для поддержки параллельных процессов с совместным использованием данных. А мои студенты в восторге от реализации sklearn.neighbors.BallTree на языке Python для поиска ближайшего соседа.

Апплеты Java из коллекции Spatial Index Demos (**http://donar.umiacs.umd.edu/quadtree/**) иллюстрируют различные варианты kd-деревьев. Алгоритмы, реализуемые в апплетах, рассматриваются в книге [Sam06]. Программа KDTREE 2 содержит реализации kd-деревьев на C++ и FORTRAN 95 для эффективного поиска ближайшего соседа в многомерных пространствах. Подробности см. по адресу **http://arxiv.org/abs/physics/0408067**.

В *разд. 20.4* представлена информация о полной коллекции реализаций алгоритмов построения диаграмм Вороного. В частности, библиотеки CGAL (**www.cgal.org**) и LEDA (см. *разд. 22.1.1*) содержат реализации на языке C++ алгоритмов построения диаграмм Вороного, а также алгоритмов выяснения местоположения точек на плоскости для их эффективного использования при поиске ближайшего соседа.

### Примечания

В последние двадцать лет в области аппроксимирующего поиска ближайшего соседа в многомерных пространствах велась активная деятельность. Недавно полученные результаты этой деятельности предоставляют общее основание для широкого класса метрик расстояний (см. [ANN+18]). В работах [AI08] и [AIR18] приведен отличный обзор алгоритмов, основанных на LSH-хешировании и методе случайных отображений для аппроксимирующего поиска ближайшего соседа в многомерных пространствах. Как теоретические, так и экспериментальные результаты (см. [ML14], [WSSJ14] и [BM0l] соответственно) указывают на то, что эти методы обладают довольно высокой точностью.

Несколько иную теоретическую основу имеет программа ANN для аппроксимирующего поиска ближайшего соседа (см. [AM93, AMN+98]). На основе набора данных создается структура разреженного взвешенного графа, после чего поиск ближайшего соседа осуществляется выполнением «жадного» обхода от произвольной точки до точки запроса. Ближайшая точка, найденная в результате нескольких таких попыток, объявляется ближайшим соседом. Подобные структуры данных оказываются полезными и при решении других задач в многомерных пространствах. Предметом пятых соревнований DIMACS как раз и был поиск ближайшего соседа (см. [GJM02]).

Самым лучшим справочником по kd-деревьям и другим пространственным структурам данных является книга [Sam06]. В ней подробно рассмотрены все основные (и многие второстепенные) варианты этих структур. Книга [Sam05] представляет собой краткий обзор той же темы. Метод случайных смещений точки запроса представлен в работе [Pan06].

Хорошие описания задачи поиска пары ближайших точек на плоскости (см. [BS76]) можно найти в книгах [CLRS09] и [Man89]. Вместо простого выбора точек из триангуляции Делоне в этих алгоритмах применяется метод «разделяй и властвуй».

### Родственные задачи

Kd-деревья (см. *разд. 15.6*), диаграммы Вороного (см. *разд. 20.4*), поиск в области (см. *разд. 20.6*).

## 20.6. Поиск в области

**Вход.** Множество $S$ из $n$ точек в $d$ измерениях и область $Q$.

**Задача.** Выяснить, какие точки множества $S$ находятся в области $Q$ (рис. 20.6).

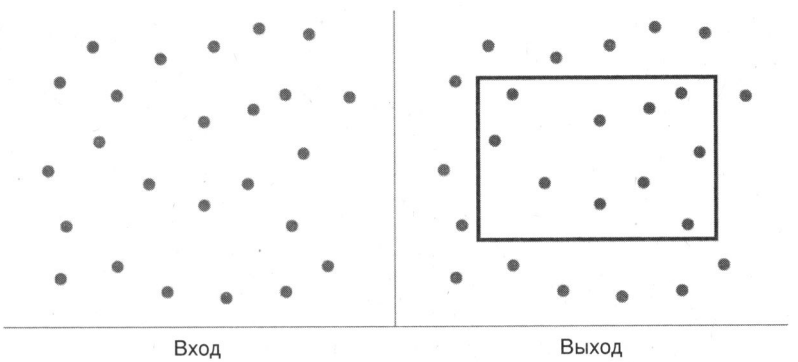



**Рис. 20.6.** Поиск в области

### Обсуждение

Задачи поиска в области (диапазонного поиска) возникают в приложениях баз данных и географических информационных систем. Объекты базы данных, имеющие $d$ числовых полей (и описывающие, например, людей по росту, весу, уровню дохода и т. д.), можно представить в виде точек в $d$-мерном пространстве. *Запрос поиска в области* (диапазонный запрос) описывает все точки (или определенное количество точек), которые мы хотим найти в заданной области пространства. Например, запрос на поиск всех людей с годовым доходом от 0 до 20 000 долларов, ростом между 185 и 215 см и весом от 25 до 68 кг определяет область, содержащую тощих людей с тощим кошельком.

Уровень трудности поиска в области зависит от нескольких факторов.

- *Велико ли количество выполняемых запросов?*

  Самый простой подход — проверка каждой из $n$ точек на попадание в многоугольник $Q$. Этот метод прекрасно работает при небольшом количестве запросов. Алгоритмы для проверки вхождения точки в заданный многоугольник рассматриваются в *разд. 20.7*.

- *Какова форма многоугольника?*

  Легче всего выполнять поиск в прямоугольниках, параллельных осям координат, поскольку проверка точки на вхождение в такую область сводится к проверке попадания каждой координаты в требуемый диапазон. Пример *поиска в ортогональной области* показан на рис. 20.6, *справа*.

  При поиске в невыпуклом многоугольнике полезно разбить его на выпуклые области или, что еще лучше, треугольники и выполнить поиск в каждой из полученных областей. Этот метод хорошо работает, потому что проверка точки на вхождение в выпуклый многоугольник является достаточно простой и быстровыполнимой задачей. Алгоритмы разбиения невыпуклого многоугольника на несколько выпуклых многоугольников рассматриваются в *разд. 20.11*.

- *Какова размерность пространства?*

  Общий подход к поиску в области состоит в создании kd-дерева для входного набора точек (см. *разд. 15.6*). Затем выполняется обход в глубину этого kd-дерева, причем каждый узел дерева разворачивается только тогда, когда соответствующий прямоугольник пересекает область запроса. В случае очень больших или смещенных областей запроса может потребоваться выполнить обход всего дерева, но, вообще говоря, kd-деревья дают эффективное решение. Для двумерного пространства известны алгоритмы с более высокой производительностью в наихудшем случае, так что на плоскости производительность kd-деревьев должна быть вполне приемлемой. А в многомерных пространствах поиска они являются единственным возможным решением задачи.

- *Возможны ли операции вставки и удаления?*

  Красивый и практичный подход к решению задачи поиска в области и ряда других геометрических задач поиска основан на триангуляциях Делоне. В триангуляции Делоне ребра соединяют каждую точку $p$ с близлежащими точками. Для поиска

в области мы сначала выясняем местоположение точки на плоскости (см. *разд. 20.7*) чтобы быстро найти треугольник в интересующей нас области. Потом мы выполняем поиск в глубину вокруг какой-либо вершины этого треугольника, разрежая пространство поиска в каждом случае, когда посещаем точку, расположенную слишком далеко, чтобы иметь неоткрытых соседей, представляющих для нас интерес. Эта процедура должна быть эффективной, поскольку общее количество посещенных точек приблизительно пропорционально количеству точек в области запроса.

Одним из достоинств такого подхода является легкость, с которой можно откорректировать триангуляцию Делоне после вставки или удаления точки, используя для этого операции замены ребер. Подробности см. в *разд. 20.3*.

- *Можно ли ограничиться подсчетом количества точек в области или требуется идентифицировать их?*

Во многих случаях достаточно только подсчитать количество искомых точек в области, а не возвращать сами точки. Например, из базы данных, упомянутой в начале этого раздела, мы можем узнать, каких людей больше: худых и бедных или толстых и богатых. Часто возникает необходимость найти самую плотную или самую разреженную область в пространстве поиска, и эту задачу можно решить с помощью подсчета точек в ней.

Хорошая структура данных для эффективного поиска ответов на составные запросы поиска в области основана на упорядочении набора точек по признаку доминирования. Говорят, что точка *x доминирует* над точкой *y*, если точка *y* располагается снизу и слева от точки *x*. Пусть $D(p)$ является функцией, которая подсчитывает количество точек во множестве $S$, над которыми доминирует точка $p$. Количество точек $m$ в ортогональном прямоугольнике, определяемом координатами $x_{\min} \le x \le x_{\max}$ и $y_{\min} \le y \le y_{\max}$, вычисляется по такой формуле:

$$m = D(x_{\max}, y_{\max}) - D(x_{\max}, y_{\min}) - D(x_{\min}, y_{\max}) + D(x_{\min}, y_{\min}).$$

Последний ее член вносит поправку на точки нижнего левого угла, которые были вычтены дважды.

Пространство можно разделить на $n^2$ прямоугольников, проводя горизонтальную и вертикальную прямые через каждую из $n$ точек. В любом прямоугольнике набор доминируемых точек будет одинаков для каждой точки, поэтому их можно вычислить заранее для нижнего левого угла каждого прямоугольника, сохранить и возвращать в ответ на запрос для любой точки в этом прямоугольнике. Таким образом, поиск в области сводится к двоичному поиску и выполняется за время $O(\lg n)$. Эта структура данных имеет квадратичную сложность по памяти, что весьма затратно. Но эту идею можно использовать с kd-деревьями, чтобы создать структуру, расходующую память более эффективно.

### Реализации

Библиотеки CGAL (**www.cgal.org**) и LEDA (см. *разд. 22.1.1*) используют динамическую структуру данных на основе триангуляции Делоне для поддержки круглых, треугольных и ортогональных областей поиска. Обе библиотеки также содержат реализации древовидных структур, которые поддерживают поиск в ортогональных областях за

время $O(k + \lg^2 n)$, где $n$ — сложность разбиения; $k$ — количество точек в прямоугольной области.

Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Эти реализации дают хорошие результаты для поиска среди сотен тысяч точек в пространствах, имеющих до 20 измерений. Библиотека позволяет выполнять запросы поиска ближайшего соседа постоянного радиуса по всем $L_p$-нормам расстояний, которые можно использовать для аппроксимации запросов поиска круглых и прямоугольных областей в нормах $L_1$ и $L_2$ соответственно. Библиотеку ANN можно загрузить по адресу **https://www.cs.umd.edu/~mount/ANN/**.

Компания Spotify разработала библиотеку Annoy на языке C++ с привязкой для языка Python (**https://github.com/spotify/annoy**). Библиотека основана на деревьях случайных проекций и предназначена для поддержки параллельных процессов с совместным использованием данных. Задача поиска ближайшего соседа тесно связана с задачей кругового поиска, поскольку круг наибольшего диаметра вокруг точки определяет ее ближайшего соседа.

#### *Примечания*

Хорошие описания структур данных с временем исполнения $O(\lg n + k)$ в наихудшем случае для поиска в ортогональных областях (см. [Wil85]) и kd-деревьев представлены в книгах [dBvKOS08] и [PS85]. В наихудшем случае производительность программ, использующих эти деревья, может быть очень низкой. Так, в работе [LW77] описывается двумерный экземпляр запроса, для которого потребовалось время $O\sqrt{n}$, чтобы выяснить, что прямоугольник пустой. Модель Word-RAM предоставляет более быстрые структуры для поиска подсчета (см. [CLP11] и [CW16]). В работе [SB19] приведены экспериментальные результаты как для последовательных, так и для параллельных алгоритмов поиска диапазона.

Задача значительно усложняется при поиске в неортогональных областях — т. е. в областях, не являющихся прямоугольниками со сторонами, параллельными осям координат. Поиск пересечений полуплоскостей имеет сложность по времени $O(\lg n)$, а по памяти — линейную (см. [CGL85]). В случае поиска в простых областях (таких как треугольник) нижние границы препятствуют созданию структур, эффективных для наихудших случаев. Обсуждение этой темы и обзор посвященных ей работ представлены в [Aga18].

### Родственные задачи

Kd-деревья (см. *разд. 15.6*), выяснение местоположения точки (см. *разд. 20.7*).

## 20.7. Местоположение точки

**Вход.** Плоскость, разбитая на многоугольники, и точка $q$.

**Задача.** Выяснить, какая область плоскости содержит точку $q$ (рис. 20.7).

### Обсуждение

Выяснение местоположения точки на плоскости является фундаментальной подзадачей вычислительной геометрии, обычно возникающей в качестве составной части решения

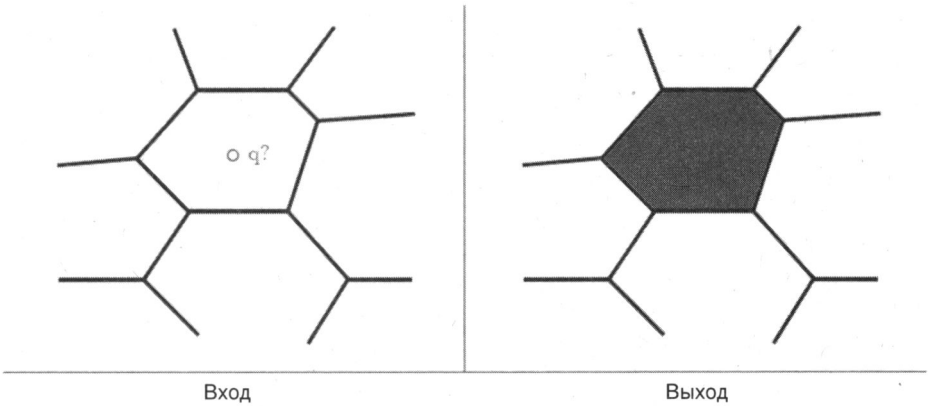



**Рис. 20.7.** Выяснение местоположения точки

больших геометрических задач. В типичной полицейской диспетчерской службе город разбивается на несколько участков (районов). Имея карту таких участков и точку запроса (место преступления), диспетчер должен найти участок, содержащий нужную точку. Это и есть задача выяснения местоположения точки на плоскости. Возможны следующие варианты задачи:

- *находится ли искомая точка внутри многоугольника P?*

  В самом простом варианте задачи выяснения местоположения точки на плоскости участвуют только две области: внутри и вне многоугольника $P$ — и требуется выяснить, в какой из них находится искомая точка. Для невыпуклых многоугольников, имеющих много узких выступов, поиск ответа может оказаться очень трудным. Процедура получения решения состоит в следующем. Из точки проводим отрезок, заканчивающийся далеко за пределами многоугольника, и подсчитываем количество ребер многоугольника $P$, пересеченных таким отрезком. Точка будет находиться внутри многоугольника тогда и только тогда, когда это число нечетное. Надо ли подсчитывать, когда отрезок проходит через вершину, а не ребро, очевидно из контекста, поскольку нас интересует количество пересечений границы многоугольника $P$. Проверка всех $n$ сторон многоугольника на пересечение с отрезком занимает время $O(n)$. Для выпуклых многоугольников существуют более быстрые алгоритмы на основе двоичного поиска со временем исполнения $O(\lg n)$;

- *сколько потребуется запросов?*

  Выяснение местоположения точки внутри многоугольника можно было бы производить отдельно для каждой области в имеющемся разбиении. Однако это будет неэкономично в случае выполнения большого количества таких запросов. Намного лучшее решение — создать поверх этого разбиения решеточную или древовидную структуру данных, позволяющую быстро попасть в нужную область. Такие структуры подробно рассматриваются далее в этом разделе;

- *какова сложность областей разбиения?*

  Если области, на которые разбита плоскость, не являются выпуклыми многоугольниками, потребуются достаточно сложные проверки на вхождение точки в область.

Но выполнив предварительную триангуляцию всех многоугольников разбиения, мы сведем проверку на вхождение точки в область к проверке на вхождение точки в треугольник. Такие проверки можно сделать очень быстрыми и простыми за счет небольших затрат на сохранение имени многоугольника для каждого треугольника. Дополнительная выгода от подобной триангуляции заключается в том, что чем меньше размер областей разбиения, тем выше производительность решеточных или древовидных суперструктур. Однако при триангуляции следует соблюдать осторожность, чтобы избежать получения треугольников с очень острыми углами (см. *разд. 20.3*);

- *насколько регулярны размер и форма областей разбиения?*

  Когда все треугольники имеют приблизительно одинаковый размер и форму, можно использовать самый простой способ выяснения местоположения точки, при котором на область разбиения накладывается решетка размером $k \times k$ из горизонтальных и вертикальных прямых, идущих через одинаковые интервалы. Для каждой из $k^2$ прямоугольных ячеек поддерживается список всех областей разбиения, которые хотя бы частично покрываются этой ячейкой. Для выяснения местоположения точки в таком сеточном файле выполняется двоичный поиск или поиск в хеш-таблице, чтобы найти ячейку, содержащую точку $q$, а потом выполняется поиск в каждой области, покрываемой ячейкой, чтобы выяснить, какая из них содержит искомую точку.

  Производительность таких сеточных файлов может быть очень хорошей при условии, что каждая треугольная область перекрывает лишь сравнительно небольшое количество прямоугольников (тем самым минимизируя потребности в памяти), а каждый прямоугольник перекрывает лишь небольшое количество треугольников (тем самым минимизируя время поиска). Производительность этого метода зависит от регулярности областей разбиения. Определенной гибкости можно добиться, располагая горизонтальные линии решетки не на одинаковых расстояниях друг от друга, а в зависимости от расположения областей разбиения. Рассматриваемый далее *метод полос* является развитием этой идеи и гарантирует эффективную временную сложность за счет квадратичной сложности по памяти;

- *какова размерность пространства?*

  Для трех и более измерений наиболее подходящим методом выяснения местоположения точки почти наверняка будет использование kd-дерева. Эти деревья могут оказаться даже лучшим решением, когда плоскость разбита на области, слишком нерегулярные, чтобы можно было применить сеточные файлы.

  Kd-деревья (см. *разд. 15.6*) иерархически разбивают пространство на прямоугольные блоки. В каждом узле дерева текущий прямоугольник разбивается на небольшое количество меньших прямоугольников (обычно 2, 4 или $2^d$ для $d$ измерений). Прямоугольник, принадлежащий листу, помечается списком областей, содержащихся в нем хотя бы частично. Процесс выяснения местоположения точки начинается с корня дерева и движется вниз по дереву через узел-потомок, чей прямоугольник содержит искомую точку $q$. Когда поиск доходит до листа, проверяется каждая содержащаяся в нем подходящая область, чтобы выяснить, какая из них включает в себя точку $q$. Так же как и в случае с сеточными файлами, мы надеемся, что каж-

дый лист будет содержать небольшое количество областей и что каждая область не пересекает слишком много листовых прямоугольников;

- *близко ли целевая ячейка?*

  Простым методом выяснения местоположения точки, который хорошо работает в пространствах, имеющих гораздо больше двух измерений, можно признать обход. Начинаем поиск с произвольной точки $p$ в произвольной ячейке, расположенной (предположительно) недалеко от точки $q$. Строим луч от $p$ к $q$ и находим сторону (грань) ячейки, пересекаемую этим лучом. В триангуляционных разбиениях такие запросы (называемые *лучевыми запросами*) выполняются за постоянное время.

  Переходя к следующей ячейке сквозь эту грань, мы еще на шаг приблизимся к цели. Для достаточно регулярных $d$-мерных разбиений ожидаемая длина пути будет $O(n^{1/d})$, но линейной в наихудшем случае.

Самым простым алгоритмом, гарантирующим время поиска $O(\lg n)$ в наихудшем случае, считается *метод полос*, в котором через каждую вершину проводятся горизонтальные прямые, определяющие $n + 1$ полос между ними. Горизонтальную полосу, содержащую конкретную точку $q$, можно найти, выполнив двоичный поиск по $y$-координате этой точки. Область в полосе, содержащую точку $q$, можно найти с помощью двоичного поиска по ребрам, пересекающим эту полосу. Некоторая трудность заключается в том, что для каждой полосы необходимо сопровождать двоичное дерево поиска, что в наихудшем случае дает сложность по памяти, равную $O(n^2)$. Метод, расходующий память более эффективно, основан на создании иерархической триангуляционной структуры на областях разбиения. Он тоже обеспечивает сложность по времени, равную $O(\lg n)$, и упомянут в *подразделе «Примечания»*.

Эффективные методы поиска точки для наихудших случаев либо требуют большого объема памяти, либо очень сложны для реализации. Но для большинства приложений обычно рекомендуется использовать kd-деревья.

### Реализации

Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *разд. 22.1.1*) содержат реализации на языке C++ самых разных алгоритмов разбиения плоскости. В библиотеке CGAL отдается предпочтение стратегии «переход и обход» (jump-and-walk), хотя также представлена реализация алгоритма поиска с логарифмической сложностью для наихудшего случая. Библиотека LEDA включает программы выяснения местоположения точки с использованием частично устойчивых (partially persistent) деревьев поиска с ожидаемым временем исполнения $O(\lg n)$.

Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Программы из этой библиотеки можно использовать для быстрого поиска точки на границе ближайшей ячейки, с которой следует начинать обход. Загрузить библиотеку можно по адресу **https://www.cs.umd.edu/~mount/ANN/**.

Пакет Arrange на языке C предназначен для размещения многоугольников на плоскости или на сфере. Многоугольники могут быть вырожденными, и в таком случае работа сводится к размещению отрезков. В пакете используется рандомизированный инкрементальный алгоритм и поддерживается эффективное выяснение местоположения точ-

ки в разбиении. Пакет Arrange разработан Майклом Голдвассером (Michael Goldwasser) и доступен на веб-сайте **http://euler.slu.edu/~goldwasser/publications**.

В книгах [O'R01] и [SR03] представлены процедуры проверки расположения точки в простом многоугольнике.

#### *Примечания*

Отличный обзор последних достижений в области решения задачи выяснения местоположения точки, как теоретических, так и практических, приведен в [Sno18]. Очень подробные описания детерминистических структур данных, используемых при выяснении местоположения точки на плоскости, содержатся в книгах [dBvKOS08] и [PS85].

В работе [TV01] задача выяснения местоположения точки на плоскости используется в качестве учебного примера разработки алгоритмов вычислительной геометрии на языке Java. Экспериментальное исследование алгоритмов выяснения местоположения точки на плоскости описано в работе [EKA84]. Лучшим был признан метод группирования, аналогичный методу сеточного файла. В работах [HH09] и [HKH12] рассмотрена производительность реализации алгоритма для определения местонахождения точки из библиотеки CGAL.

Элегантный метод улучшения треугольников (см. [Kir83]) заключается в создании иерархической структуры триангуляционных разбиений поверх имеющегося разбиения таким образом, что каждый треугольник на своем уровне пересекает постоянное количество треугольников следующего уровня. Так как размер каждой триангуляции составляет определенную долю от размера следующей триангуляции, то общая сложность по памяти вычисляется суммированием геометрической прогрессии и, следовательно, является линейной. Кроме этого, высота иерархической структуры равна $O(\lg n)$, что обеспечивает быстрое время исполнения запросов. Альтернативный алгоритм с такими же временными пределами изложен в работе [EGS86]. Описанный ранее метод полос разработан Добкином (Dobkin) и Липтоном (Lipton) (см. [DL76]) и представлен в книге [PS85]. Описания алгоритмов, проверяющих вхождение точки в простой многоугольник, содержатся в работах [O'R01], [PS85] и [SR03].

В последнее время наблюдается интерес к динамическим структурам данных, которые поддерживают быстрое инкрементальное обновление разбиения плоскости (после вставки и удаления ребер и вершин), а также быстрое выяснение местоположения точки. Изучение этой области можно начать с [CT92], а обновленный справочный материал можно найти в [Sno18]. Время поиска и обновления наилучших современных методов приближается к логарифмическому (см. [CN18]).

#### Родственные задачи

Kd-деревья (см. *разд. 15.6*), диаграммы Вороного (см. *разд. 20.4*), поиск ближайшего соседа (см. *разд. 20.5*).

## 20.8. Выявление пересечений

**Вход.** Множество $S$ отрезков (прямых) $l_1$, ..., $l_n$ или пара многоугольников (многогранников) $P_1$ и $P_2$.

**Задача.** Выяснить, какие отрезки пересекаются. Найти пересечение объектов $P_1$ и $P_2$ (рис. 20.8).

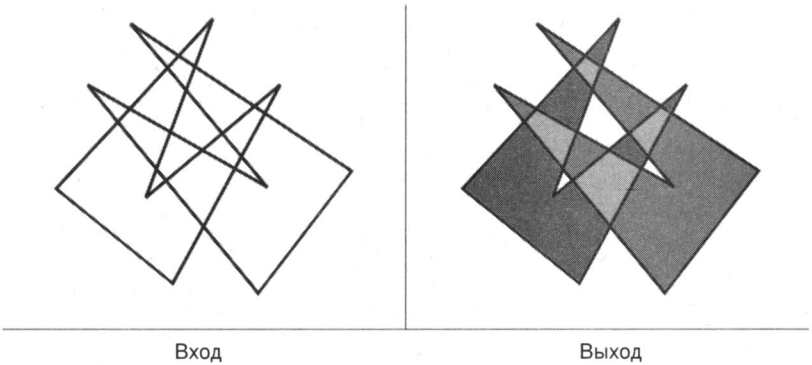



**Рис. 20.8.** Выявление пересечений

### Обсуждение

Выявление пересечения представляет собой фундаментальную задачу вычислительной геометрии, имеющую много применений. Предположим, что мы моделируем здание в виртуальной реальности компьютерной игры. Любая иллюзия реальности исчезнет, как только виртуальный персонаж пройдет сквозь стену. Чтобы обеспечить соблюдение физических ограничений, мы должны немедленно обнаруживать пересечение многогранных моделей и информировать о нем игрока или ограничивать его действия.

Еще одно применение выявления пересечений — контроль проектирования интегральных микросхем. Небольшой дефект конструкции, связанный с пересечением двух дорожек, может вызвать короткое замыкание микросхемы, и такие ошибки можно с легкостью выявить до сдачи проекта в производство, используя программы для выявления пересечения отрезков.

Вы должны ответить на следующие вопросы, возникающие в задаче выявления пересечений.

- *Нужно вычислить местоположение пересечения или достаточно лишь выявить сам факт его существования?*

  Задача просто выявления пересечения решается значительно легче, чем вычисление его местоположения, и во многих случаях этого достаточно. Для приложений виртуальной реальности важным может быть только сам факт столкновения со стеной, а не координаты точки, в которой это произойдет.

- *Нужно выявить пересечение прямых или отрезков?*

  Разница состоит в том, что любые две непараллельные прямые пересекутся в одной и только одной точке. Таким образом, все точки пересечения прямых можно вычислить за время $O(n^2)$, сравнив каждую пару прямых. Как показано в *разд. 20.15*, создание конфигурации прямых предоставляет больше информации, чем просто поиск точек пересечения.

  Поиск всех точек пересечения $n$ отрезков является значительно более трудной задачей. Даже проверка двух отрезков на пересечение оказывается не такой уж простой операцией (см. *разд. 20.1*). Можно явно проверить каждую пару отрезков и таким

образом найти все пересечения за время $O(n^2)$, но для случаев с небольшим количеством точек пересечения существуют более быстрые алгоритмы.

- *Каково ожидаемое количество точек пересечения?*

  При контроле проектирования интегральных микросхем мы ожидаем, что набор отрезков будет иметь небольшое количество точек пересечения — если они вообще найдутся. В этом случае нам нужен алгоритм, *чувствительный к выводу*, — т. е. имеющий время исполнения, пропорциональное количеству точек пересечения.

  Такие чувствительные к выводу алгоритмы для выявления пересечений отрезков уже существуют. Время исполнения самого быстрого из них равно $O(n \lg n + k)$, где $k$ — количество пересечений. Эти алгоритмы основаны на методе заметающей прямой.

- *Видна ли точка x из точки y?*

  В некоторых случаях требуется узнать, можно ли в заполненном препятствиями пространстве видеть точку $y$ из точки $x$. Эту задачу можно сформулировать в виде задачи выявления пересечения отрезков: пересекает ли отрезок между точками $x$ и $y$ какое-либо препятствие? Задачи выяснения видимости возникают при планировании перемещений роботов (см. *разд. 20.14*) и при исключении скрытых поверхностей в компьютерной графике.

- *Являются ли пересекающиеся объекты выпуклыми?*

  Существуют очень хорошие алгоритмы для выявления пересечения многоугольников. Важность при этом имеет вопрос о выпуклости многоугольников. Пересечение выпуклого $n$-угольника с выпуклым $m$-угольником можно выявить за время $O(n + m)$ с помощью алгоритма, основанного на методе заметающей прямой, описанном далее. Это возможно благодаря тому, что в результате пересечения двух выпуклых многоугольников всегда создается выпуклый многоугольник, содержащий самое большее $n + m$ вершин.

  Однако пересечение двух невыпуклых многоугольников лишено этих достоинств. Рассмотрим пересечение двух «флажков», показанное на рис. 20.8. Как видно из рисунка, пересечение невыпуклых многоугольников может быть фрагментированным.

  Задача поиска пересечения многогранников сложнее задачи поиска пересечения многоугольников, поскольку многогранники могут пересекаться даже тогда, когда их ребра не пересекаются. Примером такой ситуации может служить иголка, пронзающая внутреннюю область грани. Однако в целом аналогичные проблемы возникают при поиске пересечения как для многогранников, так и для многоугольников.

- *Перемещаются ли объекты?*

  В примере с прохождением сквозь стену в виртуальном здании неподвижные объекты не изменяются от эпизода к эпизоду. Но виртуальный персонаж перемещается, и поэтому необходимо выполнять многократный анализ фиксированной геометрической структуры.

  В таких случаях типичным подходом является аппроксимация объектов более простыми охватывающими объектами — например, параллелепипедами. Пересечение двух таких охватывающих объектов означает, что содержащиеся в них объекты

*могут* пересекаться, и для выяснения этого вопроса требуется дополнительная обработка. Но проверка на пересечение простых параллелепипедов происходит намного эффективнее, чем более сложных объектов, поэтому в случае небольшого количества пересечений этот метод выгоден. Возможны различные варианты предложенной схемы, но основная идея позволяет заметно повысить производительность в сложной обстановке.

Для эффективного поиска пересечений нескольких отрезков или пересечений и объединений двух многоугольников можно использовать алгоритмы, основанные на методе заметающей прямой. Эти алгоритмы отслеживают важные события в процессе перемещения вертикальной прямой слева направо по набору входных данных. В самом левом положении прямая не пересекает никаких объектов, но в процессе ее движения вправо происходит следующая последовательность событий:

- *вставка.*

  Найден левый конец отрезка, и этот отрезок может пересекать какой-либо другой отрезок прямой;

- *удаление.*

  Найден правый конец отрезка. Это означает, что заметающая прямая полностью прошла весь отрезок, и поэтому его можно безопасно удалить, исключая из дальнейшего рассмотрения;

- *пересечение.*

  Если для активных отрезков, пересекаемых заметающей прямой, поддерживается упорядочивание по расположению (сверху вниз), то следующее пересечение всегда должно произойти между парой соседних отрезков. После такого пересечения относительный порядок этих двух отрезков меняется на обратный.

Для слежения за процессом требуются две структуры данных. Под будущие события отводится *очередь событий* — очередь с приоритетами, упорядоченная по $x$-координате всех возможных будущих событий: вставок, удалений и пересечений. Базовые реализации очередей с приоритетами рассматриваются в *разд. 15.2*. Настоящее представлено *горизонтом* — упорядоченным списком отрезков, пересекающих текущую позицию заметающей прямой. Для сопровождения горизонта можно использовать любую словарную структуру данных — например, сбалансированное дерево.

Для поиска пересечения или объединения многоугольников изменяется обработка трех основных типов событий. Приведенный алгоритм заметающей прямой значительно упрощается для выявления пересечения пар выпуклых многоугольников, поскольку, во-первых, заметающая прямая пересекает самое большее четыре ребра многоугольников, что делает горизонт ненужным, и, во-вторых, не требуется сортировать очередь событий из-за того, что мы можем начать обработку с самой левой вершины каждого многоугольника и двигаться вправо в соответствии с естественным порядком вершин многоугольника.

Поиск пересечения или объединения любых многоугольников более сложен, но описанный здесь подход с использованием заметающей прямой справится и с этой задачей.

### Реализации

Библиотеки LEDA (см. *разд. 22.1.1*) и CGAL (**www.cgal.org**) содержат реализации на языке C++ самых разных алгоритмов для выявления пересечений отрезков и многоугольников. В частности, обе библиотеки включают реализацию алгоритма Бентли — Оттманна (Bentley — Ottmann), основанного на методе заметающей прямой (см. [BO79]), для поиска всех $k$ точек пересечения между $n$ отрезками прямых на плоскости за время $O((n + k)\lg n)$.

Стабильно работающая программа на языке C для поиска пересечения двух выпуклых многоугольников представлена в книге [O'R01]. Подробности см. в *разд. 22.1.9*.

Поиск взаимного пересечения набора полупространств является частным случаем задачи поиска пересечения выпуклых оболочек. Лучшей программой для работы с выпуклыми оболочками в многомерных пространствах считается Qhull (см. [BDH97]). Она широко используется в научных приложениях и доступна на веб-сайте **https://www.qhull.org/**.

#### *Примечания*

Отличный обзор алгоритмов поиска пересечений таких геометрических объектов, как отрезки, многоугольники и многогранники, представлен в работе [Mou18]. Книги [dBvKOS08], [CLRS09] и [PS85] содержат главы, в которых обсуждается задача поиска пересечений геометрических объектов. Подробное обсуждение частного случая поиска пересечений и объединений прямоугольников, ориентированных вдоль осей координат (задача, которая часто возникает при разработке интегральных микросхем), приведено в книге [PS85].

Алгоритм поиска пересечения отрезков с временем исполнения $O(n \lg n + k)$ рассматривается в работе [CE92]. Всесторонний обзор более простых рандомизированных алгоритмов с такими же временными границами содержится в работе [Mul94].

Алгоритмы и программное обеспечение для обнаружения столкновений обсуждаются в работе [LM04]. В работе [Wel13] подробно описаны последние структуры данных, причем особое внимание уделяется обнаружению столкновений для тактильной обратной связи. Еще одна проблема в области выявления столкновений в виде деформируемых моделей, чья форма меняется с течением времени, рассматривается в работе [BEB12]. А в работе [FK10] излагается интересная версия задачи выявления столкновений на основе идентификации дорожных перекрестков по видеоинформации или данным системы глобального позиционирования.

### Родственные задачи

Конфигурация прямых (см. *разд. 20.15*), планирование перемещений (см. *разд. 20.14*).

## 20.9. Разложение по контейнерам

**Вход.** Набор из $n$ объектов размером $d_1$, ..., $d_n$. Набор из $m$ контейнеров емкостью $c_1$, ..., $c_n$.

**Задача.** Уложить все объекты в контейнеры, задействовав наименьшее количество контейнеров (рис. 20.9).

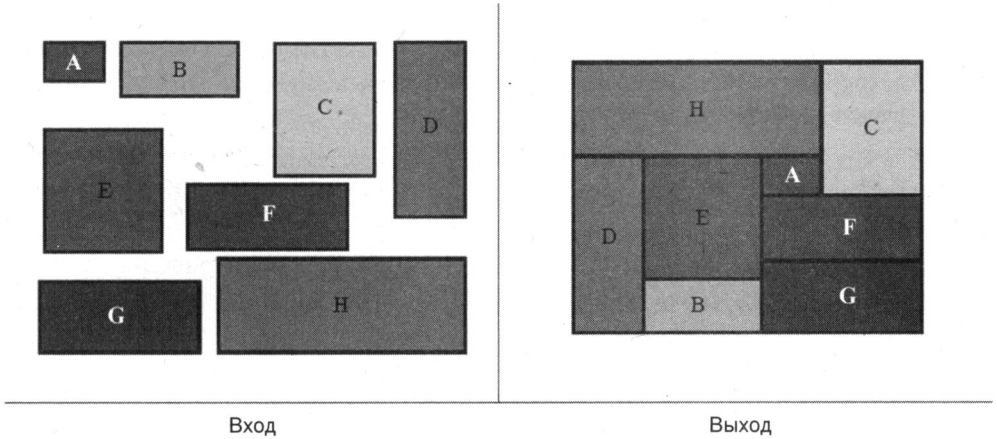



**Рис. 20.9.** Разложение по контейнерам

### Обсуждение

Задача разложения по контейнерам (корзинам) возникает во многих приложениях производства и упаковки. Для примера рассмотрим раскройку листового железа под детали или раскройку ткани для шитья одежды. Для минимизации отходов мы хотим разместить детали или выкройки так, чтобы использовать как можно меньшее количество листов железа или рулонов материи. Размещение каждой детали на листе в наилучшем для нее месте и представляет собой вариант задачи разложения по контейнерам, называемой также *задачей раскроя* (cutting stock problem). После изготовления деталей возникает другая задача — упаковки: как погрузить ящики с деталями на грузовики, чтобы минимизировать необходимое количество грузовиков.

Даже самые простые на первый взгляд задачи разложения по контейнерам являются NP-полными (см. обсуждение задачи разбиения множества целых чисел в *разд. 16.10*). Поэтому нам приходится использовать эвристические подходы, а не алгоритмы, выдающие оптимальное решение в наихудшем случае. К счастью, для большинства задач разложения по контейнерам обычно хорошо подходят сравнительно простые эвристические методы. Более того, многие приложения имеют специфические ограничивающие условия, которые не позволяют использовать сложные алгоритмы решения задачи разложения по контейнерам. Выбор эвристического метода для решения конкретной задачи будет зависеть от следующих факторов:

- *формы и размера объектов.*

  Характер задачи разложения по контейнерам в большой степени зависит от формы пакуемых объектов. Собрать пазл (картинку-головоломку) гораздо сложнее, чем уложить квадратики на прямоугольном поле. В *одномерной задаче разложения по контейнерам* размер каждого объекта указывается в виде целого числа. Эта задача эквивалентна задаче упаковки ящиков одинаковой ширины в контейнер такой же ширины и является частным случаем задачи о рюкзаке (см. *разд. 16.10*).

  Когда все объекты имеют одинаковый размер и форму, последовательное заполнение каждого ряда дает приемлемую, но не обязательно оптимальную упаковку. По-

пробуйте заполнить квадрат размером 3×3 прямоугольниками размером 2×1. Укладывая прямоугольники, ориентированные только в одном направлении, можно разместить только три из них, а если их ориентировать в двух направлениях, то поместятся четыре прямоугольника;

- *ограничения на ориентацию и размещение объектов.*

  На практике у многих упаковочных коробок обозначен верх (что предписывает определенную ориентацию коробки) или присутствует надпись «не складывать в штабель» (означающая, что эта коробка может находиться только на самом верху штабеля). Такие условия ограничивают нашу свободу действий при упаковке и увеличивают количество грузовиков, необходимых для транспортировки товаров. Большинство транспортных компаний решают эту задачу, не обращая внимания на подобные маркировки. Несомненно, любая задача становится намного легче, если не беспокоиться о последствиях;

- *статичность или динамичность задачи.*

  Известен ли нам весь набор объектов для упаковки в начале работы (статическая задача) или же мы получаем их по одному и должны укладывать их в спешке по мере поступления (динамическая задача)? Упаковку можно выполнить гораздо лучше, если есть возможность планирования. Например, имеет смысл расположить объекты в таком порядке, который позволит осуществить более эффективную упаковку (т. е. отсортировать их по убыванию размера).

В стандартных эвристических методах статического разложения по контейнерам объекты упорядочиваются по размеру или форме, а потом укладываются в контейнеры. Типичные правила выбора контейнера:

- выбираем первый или самый левый контейнер, в который может поместиться объект;
- выбираем контейнер с наибольшим свободным объемом;
- выбираем контейнер с наименьшим свободным объемом, достаточным для размещения объекта;
- выбираем случайный контейнер.

Аналитические и экспериментальные результаты показывают, что самым лучшим эвристическим методом разложения по контейнерам является «первый подходящий контейнер в порядке убывания размера объектов». Сортируем объекты по убыванию размера. Вкладываем объекты по одному в контейнер, в котором имеется достаточно места для очередного объекта. Если в контейнере больше нет места, то переходим к следующему контейнеру. В случае одномерного разложения по контейнерам при таком подходе потраченное количество контейнеров никогда не превысит действительно необходимое больше чем на 22%, причем обычно этот показатель намного лучше. Описанный метод интуитивно привлекателен, поскольку мы укладываем большие объекты в первую очередь и надеемся, что удастся уложить меньшие объекты в оставшееся место.

Этот алгоритм легко реализуется и имеет время исполнения $O(n \lg n + bn)$, где $b \leq \min(n, m)$ — количество фактически заполненных контейнеров. Здесь мы для каждого объекта выполняем перебор контейнеров за линейное время, проверяя на наличие

свободного места. Возможен более быстрый алгоритм с временем исполнения $O(n \lg n)$, использующий двоичное дерево для отслеживания свободного места, оставшегося в каждом контейнере.

Чтобы удовлетворить специфическим ограничивающим условиям, можно разнообразить порядок размещения объектов. Например, ящики с маркировкой «не складывать в штабель» следует размещать в последнюю очередь (возможно, предварительно искусственно уменьшив высоту укладки, чтобы оставить вверху достаточно свободного места), а ящики с маркировкой «верх» — в первую очередь (чтобы иметь больше свободы при укладывании на них других ящиков).

Укладывать ящики легче, чем объекты произвольной геометрической формы. Это настолько легче, что единственное общее правило укладки произвольных объектов состоит в предварительной упаковке их в индивидуальные ящики, после которой уже выполняется укладка ящиков в целевые контейнеры. Найти охватывающий прямоугольник для объекта-многоугольника не составляет труда: просто находим верхнюю, нижнюю, левую и правую касательные, идущие в заданном направлении. Выяснить ориентацию, минимизирующую площадь (объем) такого прямоугольника (ящика), намного труднее, но все-таки возможно как на плоскости, так и в трехмерном пространстве (см. [O'R85]).

В случае невыпуклых объектов большой объем полезного пространства может оказаться неиспользованным из-за наличия пустот, получившихся после помещения деталей в ящики. Одно из решений этой проблемы — найти *наибольший пустой прямоугольник* внутри каждого размещенного объекта и, если этот прямоугольник достаточно велик, поместить в него другие объекты.

### Реализации

Библиотека BPPLIB (**http://or.dei.unibo.it/library/bpplib**) содержит обширную коллекцию ресурсов для решения задачи разложения по контейнерам, включая коды, примеры задачи и справочный материал (см. [DIM18]). Если вы заинтересованы в задачах разложения по контейнерам и раскроя, эта библиотека должна быть первой в вашем списке ресурсов.

Коллекцию реализаций алгоритмов на языке FORTRAN для разных версий задачи о рюкзаке можно загрузить с веб-сайта **http://www.or.deis.unibo.it/kp.html**. Там же доступна электронная версия книги [MT90a]. Хорошо организованную коллекцию реализаций алгоритмов на языке C для решения разных видов задачи о рюкзаке и родственных ей задач — таких как разложение по контейнерам и загрузка контейнера, можно найти на веб-сайте **http://www.diku.dk/~pisinger/codes.html**.

Первым шагом упаковки объектов произвольной формы в контейнер является их размещение по индивидуальным прямоугольным контейнерам минимального объема. Код для получения аппроксимации оптимального разложения предлагается на веб-сайте **https://sarielhp.org/research/papers/00/diameter/diam_prog.html**. Этот алгоритм имеет почти линейное время исполнения (см. [BH01]).

#### Примечания

Обзоры литературы, посвященной задаче разложения по контейнерам и задаче раскроя, приведены в работах [CJCG+13], [CKPT17], [DIM16] и [WHS07]. Солидный справочник

по различным вариантам задачи о рюкзаке представляет собой книга [KPP04]. Экспериментальные результаты эвристических алгоритмов решения задачи разложения по контейнерам изложены в работах [BJLM83] и [MT87].

Существуют эффективные алгоритмы поиска наибольшего пустого прямоугольника в многоугольнике (см. [DMR97]) и наборе точек (см. [CDL86]).

Упаковка шаров — это важный и хорошо изученный частный случай разложения по контейнерам. Общеизвестна гипотеза Кеплера, касающаяся поиска наиболее плотной упаковки единичных трехмерных шаров. Справедливость этой гипотезы доказана в 1998 году Хэйлзом (Hales) и Фергюсоном (Ferguson), а само доказательство приводится в работе [Szp03]. Самым лучшим справочником по задаче упаковки шаров и родственным ей задачам является книга [CS93].

Миленкович (Milenkovic) много работал над двумерной задачей разложения по контейнерам для швейной промышленности — т. е. над минимизацией количества материала, необходимого для изготовления предметов одежды. Отчеты об этой работе содержатся в книгах [DM97] и [Mil97].

### Родственные задачи

Задача о рюкзаке (см. *разд. 16.10*), задача упаковки множества (см. *разд. 21.2*).

## 20.10. Преобразование к срединной оси

**Вход.** Многоугольник или многогранник $P$.

**Задача.** Найти «скелет» объекта $P$ — т. е. набор точек внутри объекта, имеющих более чем одну ближайшую точку на границе $P$ (рис. 20.10).

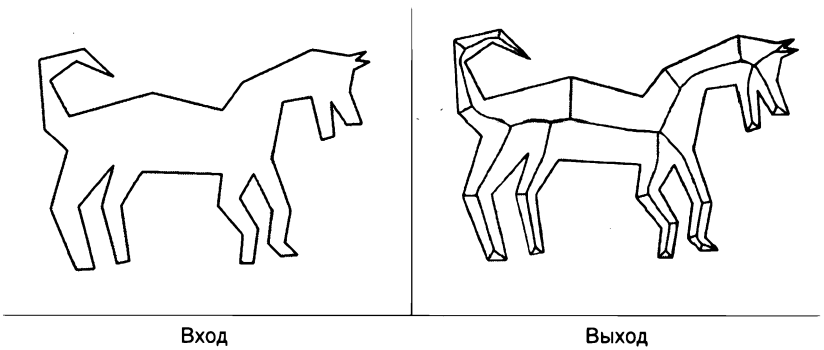



**Рис. 20.10.** Преобразование к срединной оси

### Обсуждение

Преобразование к срединной оси применимо для «утончения» многоугольников или, что эквивалентно, поиска их *скелета*. То есть поиск простого и устойчивого представления формы многоугольника. Например, «тонкие» версии букв — скажем, A и B — отражают их форму и будут в большой степени устойчивы к изменению толщины линий или добавлению декоративных элементов типа засечек. Скелет также представляет собой центральное множество точек какой-либо фигуры, как показано на рис. 20.10,

*справа*, и это его свойство может быть использовано в других приложениях — таких как восстановление формы и планирование перемещений.

Результатом преобразования к срединной оси многоугольника является дерево, что позволяет использовать динамическое программирование для вычисления «расстояния редактирования» между скелетом известной модели и скелетом неизвестного объекта. Когда два скелета достаточно близко похожи друг на друга, неизвестный объект классифицируется как экземпляр модели. Этот метод применяется в области компьютерного зрения и оптического распознавания текста. Скелетом многоугольника с отверстиями (как буквы A или B) является не дерево, а вложенный планарный граф, но с ним тоже довольно легко работать.

Существуют два разных подхода к выполнению преобразований к срединной оси — в зависимости от того, имеем ли мы на входе множество геометрических точек или растровое изображение:

- *геометрические данные.*

  Вспомним, что диаграмма Вороного для множества точек $S$ (см. *разд. 20.4*) разбивает плоскость на области вокруг каждой точки $s_i \in S$ таким образом, что все точки в области вокруг точки $s_i$ находятся ближе к ней, чем к любой другой точке из множества $S$. Аналогичным образом диаграмма Вороного для множества $L$ отрезков разбивает плоскость на области вокруг каждого отрезка $l_i \in L$ таким образом, что все точки внутри области вокруг отрезка $l_i$ находятся ближе к нему, чем к любому другому отрезку из множества $L$.

  Многоугольники определяются набором отрезков таких, что каждый отрезок $l_i$ имеет общую вершину с соседним отрезком $l_{i+1}$. Преобразование к срединной оси многоугольника $P$ сводится к получению той части диаграммы Вороного для отрезков, что находится внутри этого многоугольника. Таким образом, для утончения многоугольника подойдет любая программа создания диаграмм Вороного для отрезков.

  *Прямолинейным скелетом* называется структура, родственная срединной оси многоугольника, за исключением того обстоятельства, что биссектрисы равноудалены не от его ребер, а от вспомогательных линий этих ребер. Для выпуклых многоугольников прямолинейный скелет, преобразование к срединной оси и диаграмма Вороного идентичны, но в скелете общего вида биссектрисы могут не проходить по центру многоугольника. Прямолинейный скелет похож на результат преобразования к срединной оси, но его легче искать с помощью компьютера. В частности, все ребра прямолинейного скелета являются многоугольными;

- *изображения.*

  Оцифрованные изображения можно рассматривать как множества точек, расположенных на узлах целочисленной решетки. При этом мы можем извлечь из изображения многоугольник, представляющий объект, и обработать его с помощью ранее описанных алгоритмов вычислительной геометрии. Но внутренние вершины скелета, скорее всего, не будут располагаться на узлах/пикселах решетки. Применение алгоритмов вычислительной геометрии к задачам обработки изображений часто приводит к неудаче, потому что изображения состоят из отдельных пикселов и не являются непрерывными.

В простейшем подходе к созданию скелета пиксельного объекта применяется *метод лесного пожара* (brush fire). Представьте себе огонь, охвативший все ребра многоугольника и продвигающийся внутрь него с постоянной скоростью. Скелет образуется точками, в которых сталкиваются две или более стены огня. Такой алгоритм обходит все граничные пикселы объекта, идентифицирует вершины, принадлежащие скелету, удаляет остальную часть границы и повторяет процесс. Алгоритм прекращает работу, когда все пикселы становятся крайними, и возвращает объект толщиной всего в один или два пиксела. При правильной реализации время исполнения этого алгоритма будет линейным по отношению к количеству пикселов в изображении.

Алгоритмы, которые манипулируют непосредственно пикселами, обычно легко поддаются реализации из-за того, что в них не используются сложные структуры данных. Но при подходах, основанных на обработке пикселов, результаты получаются не вполне корректными. Например, скелет многоугольника не всегда будет деревом и не обязательно связным. Более того, точки скелета могут оказаться «не совсем» равноудаленными от двух граничных ребер. Когда вы пытаетесь применять методы непрерывной геометрии в дискретном мире, у вас нет возможности решить задачу до конца, и с этим нужно смириться.

### Реализации

Библиотека CGAL (**www.cgal.org**) предлагает пакет процедур для вычисления прямолинейного скелета многоугольника $P$. Она также содержит процедуры создания смещенных контуров, определяющих области внутри многоугольника $P$, точки которых находятся по меньшей мере на расстоянии $d$ от границ многоугольника.

Программа VRONI (см. [Hel01]) является эффективным средством создания диаграмм Вороного на плоскости для отрезков, точек и дуг. Она может с легкостью выполнять преобразования к срединной оси многоугольников, поскольку поддерживает создание диаграмм Вороного для произвольных отрезков. Программа была протестирована на огромном количестве искусственно созданных и реальных наборов данных, причем некоторые из них содержали свыше миллиона вершин. Дополнительную информацию можно найти на веб-сайте разработчика **http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html**. Другие программы создания диаграмм Вороного рассматриваются в *разд. 20.4*.

Программы для реконструкции или интерполирования облаков точек часто основаны на преобразованиях к срединной оси. Система Cocone выполняет приблизительные преобразования к срединной оси многогранной поверхности, интерполируя точки в $E^3$. Дополнительную информацию можно найти на веб-сайте разработчиков **http://www.cse.ohio-state.edu/~tamaldey/cocone.html**. Теоретические основы Cocone излагаются в работе [Dey06]. Программа Powercrust (см. [ACK01a] и [ACK01b]) выполняет дискретное приближенное преобразование к срединной оси, после чего реконструирует исходную поверхность на основе результатов этого преобразования. При достаточной плотности точек выборки алгоритм гарантированно выполняет геометрически и топологически правильную аппроксимацию поверхности. Программу можно загрузить по адресу **https://web.cs.ucdavis.edu/~amenta/powercrust.html**.

### Примечания

В книге [SP08] всесторонне рассматриваются срединные представления и соответствующие алгоритмы. Обзоры методов утончения в обработке изображений и компьютерной графике содержатся в работах [LLS92], [Ogn93], [SBdB16] и [TDS+16]. Преобразование к срединной оси впервые было использовано для изучения схожести фигур в биологии (см. [Blu67]). Вычислительная топология — новая научная область для формального анализа геометрических фигур. Дополнительную информацию см. в книге [EH10]. Обсуждение преобразований к срединной оси можно найти в книгах [dBvKOS08], [O'R01 и [Pav82].

Срединную ось произвольного $n$-угольника можно вычислить за время $O(n \lg n)$ (см. [Lee82]), хотя для выпуклых многоугольников существуют алгоритмы с линейным временем исполнения (см. [AGSS89]). Алгоритм, выполняющий преобразование к срединной оси с временем исполнения $O(n \lg n)$, представлен в работе [Kir79].

Прямолинейные скелеты обсуждаются в работе [AAAG95], а алгоритм с временем исполнения меньше квадратичного — в работе [EE99]. Интересное применение прямолинейных скелетов для создания крыш виртуальных зданий описано в работе [LD03]. Входная и выходная фигуры, приведенные на рис. 20.10, были созданы по мотивам материала из работы [dMPF09].

### Родственные задачи

Диаграммы Вороного (см. *разд. 20.4*), сумма Минковского (см. *разд. 20.16*).

## 20.11. Разбиение многоугольника на части

**Вход.** Многоугольник или многогранник $P$.

**Задача.** Разбить объект $P$ на небольшое количество простых (обычно выпуклых) частей (рис. 20.11).

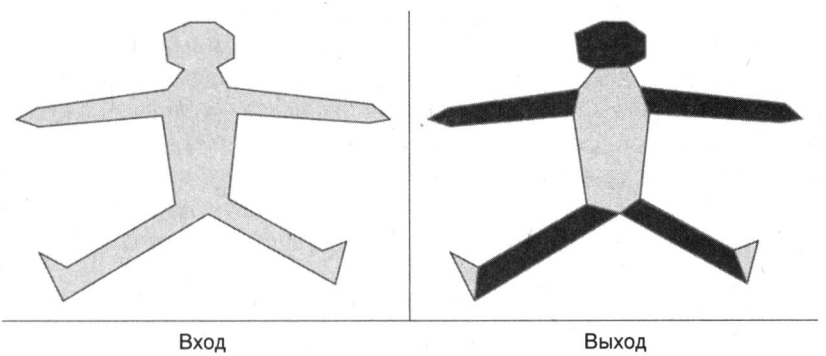



**Рис. 20.11.** Разбиение многоугольника на части

### Обсуждение

Разбиение объектов на части является важным шагом предварительной обработки во многих алгоритмах вычислительной геометрии, поскольку на выпуклых объектах геометрические задачи решаются легче, чем на невыпуклых. Работать с одним невыпуклым многоугольником сложнее, чем с несколькими выпуклыми его фрагментами.

Классической задачей разбиения многоугольника на части является подразделение страны на провинции или области и районы. Здесь обычно основное внимание уделяется равномерному распределению населения или площади каждого региона, но его форма также имеет значение. Джеррименднринг (gerrymandering) — это тонкое искусство проведения границ избирательных округов, дающее преимущество своей политической партии, обеспечив вхождение в них всех «правильных» избирателей. В результате этого избирательные округа часто принимают очень сложную форму. Чтобы предотвратить наиболее вопиющие нарушения, приняты законы, требующие, чтобы избирательные округа имели наиболее компактную форму, в идеале — опрятные выпуклые регионы.

В зависимости от конкретного приложения могут возникать разные варианты задачи разбиения многоугольника. Чтобы распознать их, ответьте на следующие вопросы.

- *Нужно ли, чтобы все фрагменты разбиения были треугольниками?*

  Триангуляция — это самая главная из задач разбиения многоугольников, поскольку в ней областями разбиения являются треугольники. Треугольник имеет всего лишь три стороны и является выпуклым, что делает его простейшим возможным многоугольником.

  Любое триангуляционное разбиение $n$-вершинного многоугольника содержит ровно $n-2$ треугольника. Поэтому триангуляция не подходит для тех случаев, когда требуется разбиение на небольшое количество выпуклых частей. Критерием «хорошей» триангуляции является не количество треугольников, а их внешний вид. Введение в триангуляцию дается в *разд. 20.3*.

- *Требуется покрытие или разбиение многоугольника?*

  Под *разбиением* многоугольника подразумевается разделение его внутренней области на части, не перекрывающие друг друга. А *покрытие* допускает перекрытие частей многоугольника. Оба способа могут оказаться полезными в разных ситуациях. Так, при декомпозиции сложного многоугольника для поиска в области (см. *разд. 20.6*) нам требуется разбиение — чтобы каждая искомая точка находилась ровно в одной области разбиения. А при декомпозиции многоугольника с целью его закрашивания вполне достаточно покрытия, поскольку двойная заливка области цветом не влечет за собой никаких проблем — при условии использования одного и того же цвета для всех частей. Мы будем рассматривать разбиение, т. к. его проще выполнить, и оно приемлемо для любого приложения, где требуется покрытие. Единственный недостаток этого подхода состоит в том, что разбиение может потребовать больше памяти, чем покрытие.

- *Можно ли добавлять дополнительные вершины?*

  Можем ли мы добавлять в многоугольник вершины Штейнера, разделяя ребра или вставляя внутренние точки? Если нет, то нам удастся лишь соединять ребрами две существующие вершины. А если да, то добавление нескольких вершин в правильном месте позволит нам получить разбиение с меньшим количеством областей, но за счет необходимости использовать более сложные алгоритмы обработки и, возможно, получения менее аккуратных результатов.

Существует простой и эффективный эвристический алгоритм Хертеля — Мельхорна для разбиения многоугольников на выпуклые области с помощью диагоналей. Алго-

ритм сначала выполняет произвольную триангуляцию многоугольника, а потом убирает все диагонали, после чего остаются только выпуклые области. Удаление диагонали создает невыпуклую область только тогда, когда в результате получается внутренний угол, превышающий 180°. Выяснить, появится ли такой угол, можно за постоянное время, рассмотрев диагонали и стороны многоугольника, окружающие удаляемую диагональ. Полученное количество выпуклых областей никогда не превысит минимально возможное их количество более чем в четыре раза.

Если минимизация количества областей разбиения не является вашей абсолютной целью, то я рекомендую использовать этот эвристический подход. Экспериментируя с разными вариантами триангуляции и разным порядком удаления диагоналей, вы, возможно, сумеете получить более качественные разбиения.

С помощью динамического программирования можно выяснить, чему равно минимальное количество диагоналей, использованных в декомпозиции многоугольников на выпуклые части. Самая простая реализация, которая отслеживает количество областей для всех $O(n^2)$ частей многоугольника, разделенных ребром, имеет время исполнения $O(n^4)$. Более быстрые алгоритмы используют более сложные структуры данных и имеют время исполнения $O(n + r^2\min(r^2, n))$, где $r$ — количество вершин с внутренним углом, превышающим 180°. Существует алгоритм с временем исполнения $O(n^3)$, который еще больше уменьшает количество областей разбиения, добавляя внутренние вершины. Но этот алгоритм сложен, и его трудно реализовать.

В другом варианте задачи многоугольник разбивается на *монотонные* области. Вершины $y$-монотонного многоугольника можно разделить на две цепочки таким образом, что любая горизонтальная линия будет пересекать любую из этих цепочек не более одного раза.

### Реализации

Многие программы триангуляции начинают работу с трапециевидного или монотонного разбиения многоугольника. Кроме этого, триангуляция является простейшим способом разбиения многоугольника на выпуклые области. В поисках отправной точки для выбора подходящего кода обратитесь к реализациям, упомянутым в *разд. 20.3*.

Библиотека CGAL (**www.cgal.org**) содержит набор процедур для разбиения многоугольников, включающий в себя реализации эвристического алгоритма Хертеля — Мельхорна для разбиения многоугольника на выпуклые области, алгоритма динамического программирования с временем исполнения $O(n^4)$ для поиска оптимального разбиения на выпуклые области и эвристического алгоритма на основе метода заметающей прямой с временем исполнения $O(n \log n)$ для разбиения на монотонные многоугольники.

Для решения задач триангуляции в двух и трех измерениях особенно хорошо подходит пакет GEOMPACK (**http://people.math.sc.edu/Burkardt/cpp_src/geompack/geompack.html**), состоящий из набора процедур на языках FORTRAN 77 и C++. Пакет позволяет выполнять триангуляцию Делоне и разбиение многоугольников и многогранников на выпуклые области, а также триангуляцию Делоне в пространстве произвольной размерности.

### Примечания

Сравнительно недавние обзоры содержатся в работах [Kei00] и [OST18]. В книге [KS85] дается отличный обзор существующего материала по разбиению и покрытию многоугольников. Описание эвристического алгоритма Хертеля — Мельхорна (см. [HM83]) можно найти в книге [O'R01]. Алгоритм динамического программирования для минимального разбиения на выпуклые области с временем исполнения $O(n + r^2\min(r^2, n))$ был предложен в работе [KS02]. Алгоритм с временем исполнения $O(r^3 + n)$ для минимизации количества выпуклых областей разбиения посредством вставки точек Штейнера представлен в книге [CD85]. В работах [LA06] и [GALL13] приведен эффективный эвристический алгоритм с временем исполнения $O(nr)$ для разложения многоугольников с внутренними пустотами на «почти выпуклые» многоугольники, причем впоследствии этот алгоритм был обобщен для работы с многогранниками.

С задачей покрытия многоугольника связана интересная *задача о картинной галерее*, в которой требуется разместить в заданном многоугольнике минимальное количество охранников таким образом, чтобы каждая внутренняя точка этого многоугольника просматривалась по крайней мере одним охранником. Это соответствует покрытию такого многоугольника минимальным количеством звездообразных многоугольников. Прекрасной книгой, в которой представлена задача о картинной галерее и ее многие варианты, является [O'R87]. Хотя, к сожалению, эта книга не переиздавалась, ее можно бесплатно загрузить по адресу **http://cs.smith.edu/~jorourke/books/ArtGalleryTheorems**. Задача о картинной галерее не является явно NP-сложной, поскольку полагается на нецелочисленные вычисления, но является сложной согласно соответствующей модели вычислений (см. [AAM18]). Результаты недавних вычислений по созданию оптимальных множеств охранников представлены в [KBFS12].

### Родственные задачи

Триангуляция (см. *разд. 20.3*), покрытие множества (см. *разд. 21.1*).

## 20.12. Упрощение многоугольников

**Вход.** Многоугольник или многогранник $p$, имеющий $n$ вершин.

**Задача.** Найти многоугольник или многогранник $p'$, имеющий только $n'$ вершин, наиболее близкий по форме к исходному объекту $p$ (рис. 20.12).

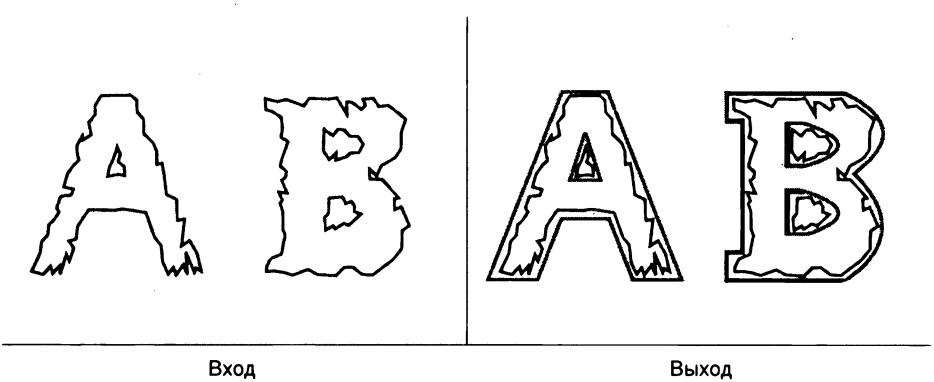



**Рис. 20.12.** Упрощение многоугольников

### Обсуждение

Задача упрощения многоугольников имеет в основном два приложения. Первое касается удаления шума из фигуры, полученной, например, сканированием изображения объекта. Упрощение ее очертания позволит устранить шум и восстановить первоначальный объект. А второе относится к сжатию данных, когда требуется избавиться от незначительных деталей большого и сложного объекта, по возможности сохранив его первоначальный вид. Это может быть особенно полезным в области компьютерной графики, поскольку меньшая модель отображается на мониторе значительно быстрее.

При решении задачи упрощения многоугольников возникает несколько вопросов.

- *Хотите ли вы получить выпуклую оболочку?*

  Выпуклая оболочка вершин объекта — это самое простое решение. Выпуклая оболочка многоугольника (см. *разд. 20.2*) удаляет все его неровности и хорошо подходит для такой задачи, как упрощение перемещений робота. Однако использование выпуклой оболочки в системе оптического распознавания текста недопустимо, поскольку вогнутости символов являются их важнейшим свойством. Например, выпуклая оболочка буквы «X» идентична выпуклой оболочке буквы «H», поскольку обе оболочки — просто прямоугольники. Кроме того, выпуклая оболочка выпуклого многоугольника никак его не упрощает.

- *Можно ли вставлять точки или разрешается только удалять их?*

  Типичной целью упрощения объекта является по возможности точное его представление с использованием заданного количества вершин. При самом простом подходе выполняются для уменьшения количества вершин локальные модификации границы. Например, если три последовательные вершины образуют небольшой треугольник, то центральную вершину можно удалить и заменить два ребра одним, не внося при этом значительных искажений в многоугольник.

  Однако подход, при котором вершины только удаляются, быстро изменит форму многоугольника до неузнаваемости. Более устойчивые эвристические методы перемещают вершины, чтобы закрыть промежутки, возникающие после удаления вершин. Такой подход типа «разделить и объединить» иногда позволяет добиться желаемого, хотя ничего не гарантирует. Получение хороших результатов гораздо вероятнее при использовании алгоритма Дугласа — Пекера, описанного далее.

- *Может ли получившийся многоугольник иметь самопересечения?*

  Серьезный недостаток инкрементальных процедур заключается в том, что они не обеспечивают получение *простых* многоугольников, т. е. многоугольников, не содержащих самопересечений. Такие «упрощенные» многоугольники могут иметь серьезные дефекты, которые вызовут проблемы на последующих этапах обработки. Если важно получить простой многоугольник, то все его отрезки нужно явно проверить на попарное пересечение (см. *разд. 20.8*).

  Подход к упрощению многоугольников, который гарантирует простую аппроксимацию, основан на поиске путей, состоящих из минимального количества отрезков. *Реберной длиной пути* между точками $s$ и $t$ называется количество отрезков в этом пути. Реберная длина прямого пути равна единице, а в общем случае она на единицу

превышает количество поворотов в пути. В помещении с препятствиями реберная длина пути между точками $s$ и $t$ определяется минимальной реберной длиной по всем путям между этими точками.

Подход к упрощению многоугольника, основанный на вычислении реберной длины, заключается в «утолщении» границы многоугольника на некоторую приемлемую величину ε (см. *разд. 20.16*), в результате чего многоугольник оказывается в своеобразном канале. Замкнутый путь с минимальной реберной длиной в этом канале представляет простейший многоугольник, границы которого не отличаются от границ исходного больше, чем на ε. Легко вычисляемая аппроксимация реберной длины сводит задачу к поиску в ширину. В канале размещается дискретный набор возможных точек поворота, после чего пары точек, находящихся в прямой видимости, соединяются ребрами.

- *Требуется очистить изображение от шума (а не упростить многоугольник)?*

  При общепринятом подходе к удалению шума из цифрового изображения на нем выполняется преобразование Фурье, отфильтровывающее высокочастотные элементы, а потом выполняется обратное преобразование, восстанавливающее изображение. Подробная информация о быстром преобразовании Фурье приведена в *разд. 16.11*.

Вместо того чтобы пытаться упростить сложный многоугольник, алгоритм Дугласа — Пекера находит для многоугольников примитивную аппроксимацию, а потом стремится улучшить ее. Для начала он выбирает две вершины $v_1$ и $v_2$ многоугольника $P$, а вырожденный многоугольник $v_1$, $v_2$ и $v_1$ рассматривает в качестве простой аппроксимации $P'$. Проходит через все вершины многоугольника $P$ и выбирает самую дальнюю от соответствующего ребра многоугольника $P'$. Вставка этой вершины добавляет треугольник к многоугольнику $P'$, минимизируя максимальное отклонение от многоугольника $P$. Такие точки можно вставлять до тех пор, пока не будет получен удовлетворительный результат. Процедура вставки $k$ точек занимает время $O(kn)$, где $|P| = n$.

В трехмерном пространстве задача упрощения становится намного труднее. Более того, задача поиска минимальной поверхности, разделяющей два многогранника, является NP-полной. В качестве эвристического алгоритма упрощения многогранников можно использовать какой-либо многомерный аналог рассматриваемых здесь плоскостных алгоритмов. Дополнительную информацию см. в *подразделе «Примечания»*.

### Реализации

Алгоритм Дугласа — Пекера достаточно прост. Реализация этого алгоритма на языке C, показывающая хорошую производительность в наихудшем случае, представлена в работе [HS94]. Программа доступна по адресу **https://www.codeproject.com/Articles/1711/A-C-implementation-of-Douglas-Peucker-Line-Approxi.**[1]

Алгоритм QSlim на основе квадратичного упрощения может довольно быстро создавать высококачественные аппроксимации триангулированных поверхностей. Реализация алгоритма доступна по адресу **http://mgarland.org/software/qslim.html**.

Еще один подход к упрощению многоугольников основан на обработке результата преобразования к срединной оси многоугольника. Преобразование к срединной оси

(см. *разд. 20.10*) создает скелет многоугольника, который можно упростить. После этого выполняется обратное преобразование, дающее в результате более простой многоугольник. Система Cocone может создавать аппроксимирующие преобразования к срединной оси многогранной поверхности, интерполируя точки в $E^3$. Дополнительную информацию можно найти на веб-сайте разработчиков **http://www.cse.ohio-state.edu/~tamaldey/cocone.html**. Теоретические основы Cocone излагаются в работе [Dey06]. Программа Powercrust (см. [ACK01a] и [ACK01b]) выполняет дискретное приближенное преобразование к срединной оси, после чего реконструирует исходную поверхность на основе результатов этого преобразования. При достаточной плотности точек выборки алгоритм гарантированно выполняет геометрически и топологически правильную аппроксимацию поверхности. Программа доступна по адресу **https://web.cs.ucdavis.edu/~amenta/powercrust.html**.

Библиотека CGAL (**www.cgal.org**) содержит процедуры для упрощения ломаных линий а также упрощения многоугольников и многогранников и поиска наименьшей охватывающей окружности или сферы.

#### *Примечания*

Алгоритм Дугласа — Пекера (см. [DP73]) составляет основу большинства схем упрощения очертаний. Быстрые реализации этого алгоритма представлены в работах [HS94] и [HS98]. В число обобщений входят алгоритмы разложения вложенных многоугольников (см. [DDS09] и [XWW11]) и сохраняющих площади упрощений (см. [BMRS16]). Подход к упрощению многоугольников, основанный на вычислении реберной длины, представлен в работе [GHMS93]. Задача упрощения очертаний становится значительно сложнее в трех измерениях.

Даже задача построения выпуклого многоугольника с минимальным количеством вершин, находящегося между двумя вложенными выпуклыми многоугольниками, является NP-сложной (см. [DJ92]), хотя существуют аппроксимирующие алгоритмы для ее решения (см. [MS95b]).

Обзор алгоритмов для упрощения очертаний представлен в работе [HG97]. Использование преобразований к срединной оси (см. *разд. 20.10*) для упрощения очертаний рассматривается в работе [TH03].

Проверку многоугольника на простоту можно выполнить за линейное время, по крайней мере теоретически, благодаря существованию линейного алгоритма триангуляции Шазеля (см. [Cha91]).

### Родственные задачи

Преобразование Фурье (см. *разд. 16.11*), выпуклая оболочка (см. *разд. 20.2*).

## 20.13. Выявление сходства фигур

**Вход.** Две фигуры: $P_1$ и $P_2$.

**Задача.** Выявить степень сходства этих фигур (рис. 20.13).

### Обсуждение

Задача выявления сходства фигур относится к области распознавания образов. Рассмотрим, например, систему оптического распознавания текста. У нас есть библиотека

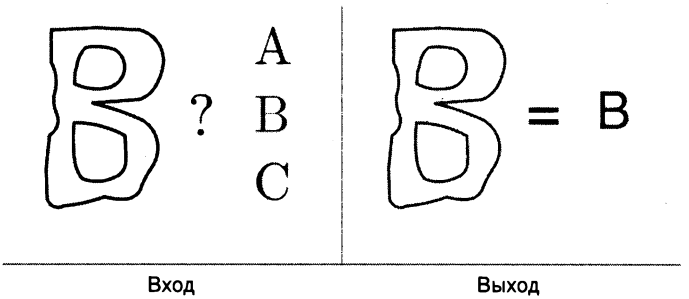



**Рис. 20.13.** Выявление сходства фигур

моделей фигур, представляющих буквы, и неизвестные фигуры, полученные в результате сканирования текста. Нам нужно идентифицировать неизвестные фигуры, сопоставив каждую из них с наиболее похожими моделями.

Задача выявления сходства плохо определена по самой своей сути, поскольку смысл слова «сходство» зависит от конкретного приложения. Из-за этого не существует единого алгоритмического подхода для решения всех задач выявления сходства фигур. Какой бы метод вы ни выбрали, вам придется потратить много времени на его настройку, чтобы добиться максимальной производительности.

Возможны следующие подходы к решению этой задачи:

- *использование расстояния Хемминга.*

  Допустим, что сравниваемые многоугольники должным образом совмещены — т. е. наложены друг друга. Расстояние Хемминга определяется площадью симметрической разности между этими двумя многоугольниками — иными словами, площадью области внутри одного многоугольника, но не внутри обоих. Когда многоугольники идентичны и должным образом выровнены, расстояние Хемминга равно нулю. Если многоугольники отличаются только небольшим шумом вдоль границы, то расстояние Хемминга будет невелико.

  Вычисление площади области симметрической разности сводится к задачам поиска пересечения и объединения двух многоугольников (см. *разд. 20.8*), с последующим вычислением площадей (см. *разд. 20.1*). Правильное выравнивание многоугольников — задача трудная. Несколько упрощается она в приложениях типа оптического распознавания текста, поскольку символы текста естественно выровнены в строках на странице. Существуют эффективные алгоритмы оптимизации наложения выпуклых многоугольников без использования вращения. Простые, но эффективные эвристические методы для решения этой задачи основаны на определении контрольных ориентиров для каждого многоугольника (таких как центр тяжести, ограничивающий прямоугольник или экстремальные вершины) с последующим сопоставлением подмножеств этих ориентиров при выравнивании многоугольников.

  Расстояние Хемминга легко вычисляется на растровых изображениях, поскольку после выравнивания изображений остается лишь сложить расстояния между соответствующими пикселами. Хотя расстояние Хемминга имеет концептуальный смысл и легко поддается реализации, оно отражает форму фигур весьма приблизительно и, скорее всего, окажется неэффективным во многих приложениях;

- *использование хаусдорфова расстояния.*

  Альтернативной метрикой схожести (после совмещения) является хаусдорфово расстояние, равное расстоянию между точкой на многоугольнике $P_1$, максимально удаленной от многоугольника $P_2$. Эта метрика несимметрична. Например, длинный и тонкий выступ многоугольника $P_1$ может значительно увеличить хаусдорфово расстояние от $P_1$ до $P_2$, даже если каждая точка $P_2$ находится поблизости от какой-либо точки $P_1$. Небольшое утолщение всей границы одной из моделей (что может случиться при наличии шума на границе) может значительно увеличить расстояние Хемминга, но при этом мало повлияет на хаусдорфово расстояние.

  Какая же из этих двух метрик лучше? Все зависит от характера вашего приложения. Кроме прочего, правильное выравнивание многоугольников может быть сопряжено с трудностями и отнимает много времени;

- *сравнение скелетов.*

  При более эффективном подходе к выявлению сходства фигур применяется утончение (см. *разд. 20.10*), позволяющее получить древоподобный скелет каждого объекта, отражающий многие характеристики исходной фигуры. После этого задача сводится к сравнению двух скелетов с использованием таких их свойств, как топология дерева и длина и наклон ребер. Это сравнение можно смоделировать в виде выяснения изоморфизма подграфов (см. *разд. 19.9*), при котором ребра считаются совпадающими при достаточной схожести их длины и наклона;

- *методы машинного обучения.*

  Наконец, можно воспользоваться каким-либо методом машинного обучения: логистической регрессией, глубинной нейронной сетью, методом опорных векторов и пр. В последние годы в этой области наблюдается серьезный прогресс, и такие приложения, как, например, распознавание лиц, выдают высокоточные результаты, несмотря на различия в позе, расположении и освещении между исследуемым изображением и искомым образцом. А ведь такие задачи решать намного труднее, чем задачи определения схожести неподвижных геометрических фигур, рассматриваемые в этом разделе.

  Применение методов машинного обучения дает успешные результаты в тех случаях, когда имеется большой объем данных для обучения, но нет ясной идеи, как использовать эти данные для решения конкретной задачи. Обычно в первую очередь нам нужно определить набор таких характеристик фигуры, которые было бы легко найти с помощью компьютера. Например, это может быть площадь, количество сторон или количество отверстий, хотя методы глубинного обучения устраняют необходимость в таком конструировании признаков. Затем программа-«черный ящик» обрабатывает обучающие данные и создает классифицирующую функцию. Эта функция принимает в качестве входа вектор заданных характеристик и возвращает меру фигуры — т. е. степень ее близости к определенной фигуре.

  Каково качество получаемых классифицирующих функций? Все зависит от особенностей вашего приложения. Методы машинного обучения требуют серьезной настройки, если вам требуется полностью использовать их потенциал. При этом одной

из проблем является интерпретируемость. Если вы не знаете, как или почему классифицирующие функции «черного ящика» принимают решения, то не сумеете распознать неверное решение, если они его выдадут. В связи с этим весьма интересен пример системы, созданной для армии и предназначенной для различения танков и автомобилей. Система прекрасно работала на тестовых изображениях, но не выдержала полевых испытаний. Наконец, кто-то сообразил, что изображения автомобилей снимались в солнечный день, а танков — в облачный, и система различала два типа объектов исключительно по присутствию облаков на заднем плане!

### Реализации

Библиотека вычислительной геометрии CGAL (**https:// www.cgal.org/**) содержит множество процедур для обнаружения и сравнения геометрических фигур, включая процедуру для вычисления хаусдорфова расстояния. Альтернативная метрика сходства многоугольников основана на угле поворота (см. [ACH+91]). Программа на языке C, использующая эту метрику, доступна по адресу **www.algorist.com**.

Существует также несколько отличных реализаций алгоритмов, основанных на методе опорных векторов, — такие как библиотека машинного обучения scikit-learn на языке Python (**https://scikit-learn.org/**), программа SVM$^{light}$ (**http://svmlight.joachims.org/**) и широко используемая и хорошо поддерживаемая библиотека LIBSVM (**https:// www.csie.ntu.edu.tw/~cjlin/libsvm/**).

#### *Примечания*

В работе [Vel01] представлено обозрение сравнения геометрических фигур с точки зрения вычислительной геометрии. Также см. обозрение [AG00]. Среди книг общей тематики по алгоритмам классификации образов можно назвать [Che15], [DHS00] и [JD88]. В работе [SBW17] описано недавнее исследование геометрических подходов к решению задачи распознавания лиц. Основным справочником по глубинному обучению является [GBC16].

Оптимальное выравнивание при сдвиге (но не при вращении) сравниваемых многоугольников из $n$ и $m$ вершин можно вычислить за время $O((n + m)\log(n + m))$ (см. [dBDK+98]). Аппроксимация оптимального наложения при сдвиге и вращении была представлена в работе [ACP+07].

Алгоритм с линейным временем исполнения для вычисления хаусдорфова расстояния между двумя выпуклыми многоугольниками приведен в работе [Ata83], а алгоритмы для общего случая — в работе [HK90].

### Родственные задачи

Изоморфизм графов (см. *разд. 19.9*), преобразование к срединной оси (см. *разд. 20.10*).

## 20.14. Планирование перемещений

**Вход.** Робот многоугольной формы, начинающий движение в точке $s$ в комнате, содержащей многоугольные препятствия, и конечная точка $t$.

**Задача.** Найти самый короткий маршрут от точки $s$ до точки $t$, не пересекающий никаких препятствий (рис. 20.14).

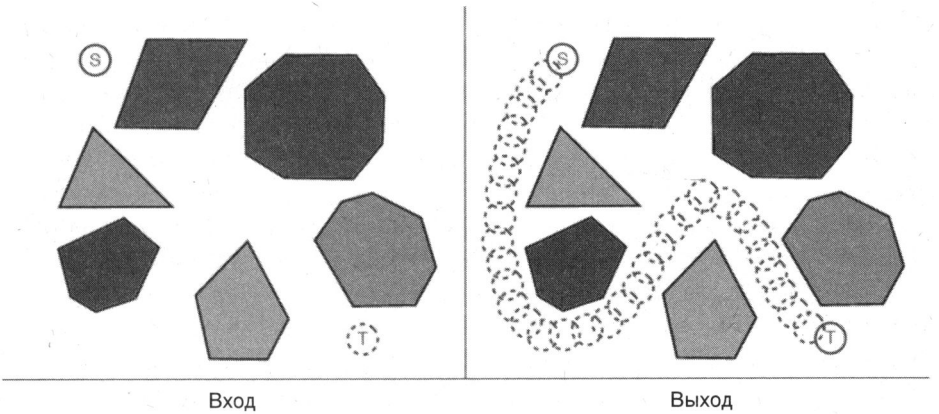



**Рис. 20.14.** Планирование перемещений

### Обсуждение

Сложность задачи планирования перемещений знакома каждому, кто пытался внести мебель в небольшую квартиру. Классическим приложением задачи планирования перемещений является разработка маршрутов роботов. Эта задача также возникает в области молекулярного докинга. Многие лекарства представляют собой небольшие молекулы, действие которых основано на связывании их с некоторой целевой молекулой. Поиск доступных участков связывания при разработке новых лекарств также является экземпляром задачи планирования перемещений.

Наконец, планирование перемещений применяется в компьютерной анимации и виртуальной реальности. Для имеющегося набора объектов и их местоположения в сценах $s_1$ и $s_2$ алгоритм планирования перемещений может создать короткую последовательность промежуточных перемещений, преобразующую сцену $s_1$ в сцену $s_2$. Эти перемещения могут заполнить промежуточные сцены между сценами $s_1$ и $s_2$, что значительно облегчит работу аниматора.

Сложность задач планирования перемещений зависит от многих факторов.

- *Является ли робот точкой?*

  Задача планирования перемещения точечного робота сводится к поиску кратчайшего пути от точки $s$ к точке $t$ без столкновений с препятствиями. Проще всего реализуется подход, который заключается в создании *графа видимости* многоугольных препятствий и точек $s$ и $t$. У этого графа видимости имеется узел для каждой вершины препятствия, а две вершины препятствия соединяются ребром тогда и только тогда, когда между ними нет ребра препятствия, мешающего им «видеть» друг друга.

  Граф видимости можно создать, перебирая всех кандидатов на ребро между $\binom{n}{2}$ пар вершин и проверяя, пересекаются ли они с каждым из $n$ ребер препятствий. Впрочем, существуют и более быстрые алгоритмы, когда каждому ребру графа видимости присваивается вес, равный его длине. Тогда кратчайший путь от точки $s$ к точке $t$ можно найти, используя алгоритм Дейкстры для поиска кратчайшего пути

(см. *разд. 18.4*). Время исполнения этого алгоритма ограничено временем, требуемым процедурой для создания графа видимости.

- *Какие действия может выполнять робот?*

  Задача планирования перемещений становится значительно труднее, когда робот — не точка, а многоугольник. Тогда ширина всех используемых для его перемещения коридоров должна быть достаточной, чтобы робот мог по ним пройти.

  Алгоритмическая сложность зависит от числа *степеней подвижности* (degrees of freedom) робота. В частности, может ли робот, кроме перемещения, совершать повороты? Есть ли у робота конечности, способные сгибаться или вращаться независимо от него, — например, рука? Каждая степень подвижности соответствует измерению пространства поиска возможных конфигураций. Дополнительная степень подвижности означает большую вероятность существования короткого маршрута, но это также делает труднее задачу поиска этого маршрута.

- *Можно ли упростить форму робота?*

  Алгоритмы планирования перемещений обычно сложны и трудоемки. Вам будет полезно все, что позволит упростить задачу. В частности, рассмотрите возможность помещения робота в охватывающую окружность. Любой маршрут для этой окружности будет также и маршрутом для находящегося внутри нее робота. Кроме того, поскольку любая ориентация окружности эквивалентна любой другой ее ориентации, то повороты при поиске пути не будут играть никакой роли. Тогда все движения будут ограничены простым перемещением.

- *Ограничены ли движения одним лишь перемещением*

  Когда повороты робота не разрешаются, можно воспользоваться методом *расширения препятствий*, чтобы свести задачу планирования перемещения робота-многоугольника к ранее решенной задаче планирования перемещения робота-точки. Для этого выбираем на роботе базисную точку и заменяем каждое препятствие его суммой Минковского с многоугольником робота (см. *разд. 20.16*). В результате получаем препятствие большего размера, включающее в себя след, оставляемый роботом, когда он движется вокруг препятствия, прикасаясь к нему. Поиск маршрута среди таких увеличенных препятствий от исходной базисной точки до цели определяет допустимый маршрут робота-многоугольника в исходном окружении.

- *Известны ли препятствия заранее?*

  До сих пор мы предполагали, что для планирования перемещений робота у нас имеется карта с расположением всех препятствий. Но это невозможно в приложениях с движущимися препятствиями. Для решения задач планирования перемещений без карты существуют две стратегии. При первом подходе мы исследуем окружение, создаем карту и на ее основе разрабатываем маршрут от начальной точки к целевой. Другой, более простой подход напоминает перемещение в тумане с помощью компаса. Идем в направлении цели до тех пор, пока наш путь не будет перекрыт препятствием. Потом движемся вокруг препятствия, пока опять не появится возможность продолжить движение в прежнем направлении. К сожалению, этот подход неприменим в достаточно сложной обстановке.

Наиболее практичный подход к решению общей задачи планирования перемещения — это произвольная выборка данных *конфигурационного пространства* робота. Конфигурационное пространство определяет набор допустимых положений робота с использованием одного измерения для каждой степени подвижности. Плоский робот, обладающий возможностью перемещения и поворота, имеет три степени подвижности, а именно $x$- и $y$-координату базисной точки на роботе и угол $\theta$ поворота относительно этой точки. Некоторые точки в этом пространстве соответствуют разрешенным позициям, а другие — препятствиям.

Набор разрешенных точек конфигурационного пространства создается случайной выборкой. Для каждой пары точек $p_1$ и $p_2$ выясняем, существует ли между ними прямой путь, не пересекающий препятствий. Таким образом строится граф, вершины которого представляют допустимые точки конфигурационного пространства, а ребра — свободные от препятствий пути между этими точками. Задача планирования перемещения теперь сводится к поиску прямого пути между начальной/конечной точкой маршрута и одной или несколькими вершинами графа и последующему поиску кратчайшего пути от исходной точки до конечной.

Существует много способов улучшения этого базового метода — например, путем добавления дополнительных вершин в области, представляющих особый интерес. Создание дорожной карты позволяет точно решать задачи, которые в противном случае выглядят очень запутанными.

### Реализации

Библиотека Open Motion Planning Library (**https://ompl.kavrakilab.org/**) включает современные алгоритмы для планирования перемещений на основе выборок с точками входа для интегрирования процедур визуализации и проверки на столкновение. Эта библиотека должна стать вашей отправной точкой в любом проекте с роботами или любом другом проекте планирования перемещений. Описание библиотеки дается в [ŞMK12].

Пакет Motion Planning Kit содержит библиотеку процедур на языке C++ и набор средств разработки планировщиков перемещений для одного и нескольких роботов. В состав набора входит SBL — быстрый вероятностный планировщик маршрутов на дорожной карте. Набор можно найти по адресу **http://robotics.stanford.edu/~mitul/mpk/**.

Библиотека вычислительной геометрии CGAL (**www.cgal.org**) содержит большое количество реализаций алгоритмов, касающихся задачи планирования перемещений, включая процедуры создания графов видимости и вычисления сумм Минковского. В книге [O'R01] предоставлена реализация алгоритма для планирования перемещений на плоскости двухшарнирного робота-манипулятора. Подробности см. в *разд. 22.1.10*.

#### Примечания

В книге [Lat91] обсуждаются практические подходы к планированию перемещений, включая описанный ранее метод случайной выборки. Две другие заслуживающие внимания книги по планированию перемещений можно загрузить бесплатно: [LaV06] — по адресу **http://planning.cs.uiuc.edu**, а [Lau98] — по адресу **https://homepages.laas.fr/~jpl/book.html**.

Задача планирования перемещений («проблема передвижения пианино») изначально изучалась Шварцем (Schwartz) и Шариром (Sharir). Разработанное ими решение создает полное пространство позиций робота, не пересекающихся с препятствиями, после чего находит кратчайший путь в соответствующем компоненте связности. Эти описания свободного от препятствий пространства очень сложны. Доклады, посвященные задаче планирования перемещений, приводятся в книге [HSS87], а в работах [KF11], [MLL16], [PČY+16] и [HSS18] дается обзор последних результатов.

Самый лучший результат для подхода, основанного на использовании свободного от препятствий пространства, приведен в [Can87], где показано, что любую задачу с $d$ степенями подвижности можно решить за время $O(n^d \lg n)$, хотя для частных случаев общей задачи планирования перемещений существуют более быстрые алгоритмы. Подход к задаче планирования перемещений, основанный на методе расширения препятствий, представлен в работе [LPW79]. Обсуждение эвристического метода «с компасом в тумане» содержится в работе [LS87].

Временная сложность алгоритмов, основанных на методе свободного от препятствий пространства, зависит от комбинаторной сложности расположения поверхностей, определяющих свободное пространство. Алгоритмы для поддержки таких компоновок представлены в *разд. 20.15*. В анализе этих компоновок часто возникают последовательности Дэвенпорта — Шинцеля. Всестороннее рассмотрение последовательностей Дэвенпорта — Шинцеля и их значимости для задачи планирования перемещений приводится в книге [SA95].

Граф видимости $n$ отрезков с $E$ парами видимых вершин можно создать за время $O(n \lg n + E)$ (см. [GM91] и [PV96]), что представляется оптимальным результатом. Алгоритм с временем исполнения $O(n \lg n)$ для поиска кратчайшего пути для робота-точки среди препятствий-многоугольников приводится в работе [HS99]. А в работе [Che85] описан алгоритм с временем исполнения $O(n^2 \lg n)$ для поиска кратчайшего пути среди препятствий-многоугольников для робота-окружности.

### Родственные задачи

Задача поиска кратчайшего пути (см. *разд. 18.4*), сумма Минковского (см. *разд. 20.16*).

## 20.15. Конфигурации прямых

**Вход.** Набор прямых $l_1, ..., l_n$.

**Задача.** Найти разбиение плоскости, определяемое набором прямых $l_1, ..., l_n$ (рис. 20.15).

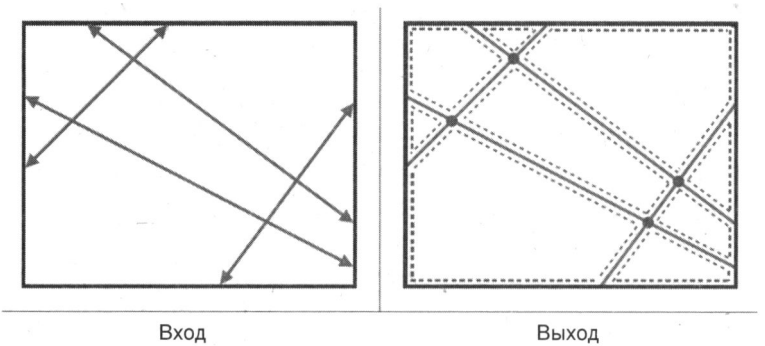



**Рис. 20.15.** Разбиение плоскости

### Обсуждение

Явное создание областей, формируемых пересечениями набора $n$ прямых, — это также одна из фундаментальных задач вычислительной геометрии наряду с выявлением пересечений, триангуляцией и другими задачами, рассматриваемыми в этой главе. Многие задачи сводятся к созданию и анализу конфигурации некоторого набора прямых. Можно привести два примера подобных задач:

- *проверка на вырожденность.*

  Для заданного набора из $n$ прямых на плоскости выяснить, проходят ли какие-либо три линии через одну и ту же точку. Проверка всех таких триад методом исчерпывающего перебора займет время $O(n^3)$. В качестве альтернативы мы можем создать конфигурацию прямых и рассмотреть каждую вершину, чтобы явно подсчитать ее степень, причем все это делается за квадратичное время;

- *удовлетворение максимальному количеству линейных ограничений.*

  Допустим, что у нас имеется набор линейных ограничений, каждое в виде $y \leq a_i x + b_i$. Нам нужно найти точку на плоскости, которая удовлетворяет самому большому количеству таких ограничений. Для решения этой задачи сначала создаем конфигурацию прямых. Все точки в любой области (или ячейке), образуемой пересекающимися линиями, будут удовлетворять одному и тому же набору ограничений, поэтому, чтобы найти глобальный максимум, достаточно проверить только одну точку в каждой ячейке.

При разработке алгоритмов бывает полезно формулировать геометрические задачи в терминах свойств конфигурации. К сожалению, следует признать, что на практике конфигурации не пользуются такой популярностью, как можно было бы предполагать. Основной причиной является то обстоятельство, что для правильного применения конфигураций требуется глубокий уровень знаний. Библиотека вычислительной геометрии CGAL предоставляет общую и достаточно устойчивую реализацию, оправдывающую усилия, чтобы разобраться в том, как использовать конфигурации. Столкнувшись с задачей конфигурации прямых, постарайтесь ответить на следующие вопросы.

- *Какой метод лучше всего подходит для создания конфигураций прямых?*

  Для этой цели используются инкрементальные алгоритмы. Начинаем с конфигурации, состоящей из одной или двух линий, и добавляем в нее по одной новые прямые, получая конфигурации все большего размера. Чтобы добавить в конфигурацию новую прямую, начинаем с самой левой ячейки, содержащей эту прямую, и идем по конфигурации вправо, перемещаясь от текущей ячейки к смежной и разбивая на две части каждую ячейку новой прямой.

- *Каким будет размер создаваемой конфигурации?*

  Согласно *теореме о зоне* $k$-я добавленная прямая пересекает $k$ ячеек конфигурации, причем $O(k)$ ребер образуют границы таких ячеек. Это означает, что мы можем перебрать все ребра каждой ячейки, обнаруживаемой в процессе добавления прямых, будучи уверенными, что общий объем работы, выполненный при добавлении прямой в конфигурацию, окажется линейным. Поэтому общее время добавления всех $n$ прямых для создания полной конфигурации составит $O(n^2)$.

- *Какова цель создания конфигурации?*

  Часто требуется найти ячейку той или иной конфигурации, содержащую заданную точку $q$. Это задача выяснения местоположения точки (см. *разд. 20.7*). А для какой-либо конфигурации прямых или отрезков часто требуется вычислить все точки пересечения этих прямых. Задача выявления пересечений обсуждается в *разд. 20.8*.

- *Входные данные — это набор точек, а не прямых?*

  Хотя прямые и точки представляют собой разные геометрические объекты, они могут заменять друг друга. Используя *преобразования двойственности*, можно преобразовать прямую $L$ в точку $p$ и наоборот:

  $$L : y = 2ax - b \leftrightarrow p: (a, b).$$

  Важность двойственности состоит в том, что мы теперь можем применять конфигурации прямых в задачах на точках, нередко получая неожиданные результаты.

  Рассмотрим пример. Имеется множество $n$ точек и требуется узнать, не располагаются ли какие-либо три из этих точек на одной и той же прямой. Эта задача похожа на задачу проверки на вырожденность, рассмотренную ранее. Но в действительности, это *та же самая задача*, только точки и прямые в ней поменялись ролями. Мы можем выполнить преобразование двойственности точек в прямые, как описано ранее, а потом выполнить поиск вершины, через которую проходят три прямые. Преобразование двойственности этой вершины определяет прямую, на которой располагаются три исходные вершины.

Часто требуется перебрать все многоугольники существующей конфигурации только один раз. Такие алгоритмы называются *алгоритмами заметающей прямой* и рассматриваются в *разд. 20.8*. Базовая процедура этих алгоритмов состоит в упорядочивании точек пересечения по $x$-координате и выполнении обхода слева направо с сохранением информации, обнаруженной в процессе обхода.

### Реализации

Библиотека CGAL (**www.cgal.org**) содержит пакет общих процедур для работы с конфигурациями кривых (а не только прямых) на плоскости. Эта библиотека должна стать отправной точкой для любого проекта, в котором используются конфигурации. Наилучшее руководство по конфигурациям CGAL — недавно изданная книга [FHW12].

Устойчивый код на языке C++ для создания и топологического заметания конфигураций можно найти на веб-сайте **http://www.cs.tufts.edu/research/geometry/other/sweep**. В библиотеке CGAL предоставлено расширение топологического заметания для работы с комплексом видимости набора попарно непересекающихся выпуклых плоских множеств.

Пакет Arrange на языке C предназначен для размещения многоугольников на плоскости или на сфере. Многоугольники могут быть вырожденными, и тогда работа сводится к размещению отрезков. В пакете используется рандомизированный инкрементальный алгоритм и поддерживается эффективное выяснение местоположения точки в разбиении. Пакет Arrange разработан Майклом Голдвассером (Michael Goldwasser) и доступен на веб-сайте **http://euler.slu.edu/~goldwasser/publications**.

### Примечания

Подробное изложение комбинаторной теории конфигураций и соответствующие алгоритмы представлены в книге [Ede87], которую следует признать основным справочником для каждого, кто серьезно интересуется этой темой. Свежие обзоры комбинаторных и алгоритмических результатов приводятся в работах [AS00] и [HS18]. Обсуждение принципов создания конфигураций можно найти в книгах [dBvKOS08] и [O'R01]. Вопросы реализации процедур библиотеки CGAL рассматриваются в работах [HH00] и [FWH04].

Конфигурации естественным образом обобщаются для случая многомерных пространств. В трех измерениях разбиение пространства определяется плоскостями, а в многомерных пространствах — гиперплоскостями. *Теорема о зоне* утверждает, что сложность любой конфигурации из $n$ $d$-мерных гиперплоскостей равна $O(n^d)$, а любая одиночная гиперплоскость пересекает ячейки со сложностью $O(n^{d-1})$. Это дает основание для алгоритма инкрементального создания конфигураций. Обход вдоль границы ячейки с целью поиска следующей ячейки, пересекаемой гиперплоскостью, занимает время, пропорциональное количеству ячеек, создаваемых добавлением гиперплоскости.

Теорема о зоне имеет несколько запутанную историю. Первоначальные ее доказательства оказались ошибочными для случая многомерных пространств. Обсуждение теоремы и верное ее доказательство приведены в работе [ESS93]. Теория последовательностей Дэвенпорта — Шинцеля тесно связана с изучением конфигураций (см. [SA95]).

Простой алгоритм заметания конфигурации прямых сортирует $n^2$ точек пересечения по $x$-координате и поэтому занимает время $O(n^2 \lg n)$. Использование топологического заметания (см. [EG89] и [EG91]) избавляет от необходимости в сортировке, вследствие чего обход конфигурации выполняется за линейное время. Этот алгоритм легко поддается реализации, и его можно задействовать, чтобы ускорить работу многих алгоритмов, в основе которых лежит метод заметающей прямой. Устойчивая реализация и экспериментальные результаты применения алгоритма представлены в работе [RSS02].

### Родственные задачи

Выявление пересечений (см. *разд. 29.8*), выяснение местоположения точки (см. *разд. 20.7*).

## 20.16. Сумма Минковского

**Вход.** Наборы точек или многоугольники $A$ и $B$, содержащие $n$ и $m$ вершин соответственно.

**Задача.** Найти сумму Минковского: $A + B = \{x + y \mid x \in A, y \in B\}$ (рис. 20.16).

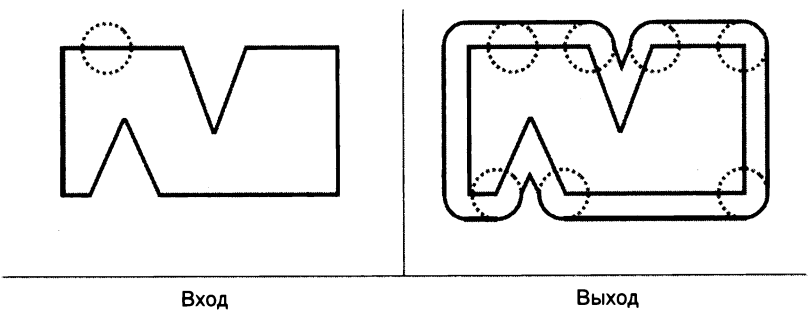



**Рис. 20.16.** Сумма Минковского

### Обсуждение

Вычисление суммы Минковского — полезная геометрическая операция, с помощью которой можно увеличивать объекты, чтобы они удовлетворяли определенным требованиям. Например, в известном подходе к планированию перемещений роботов многоугольной формы в комнате с препятствиями многоугольной формы (см. *разд. 20.14*) размер каждого из препятствий увеличивается на значение суммы Минковского препятствия и робота, в результате чего эта задача сводится к более простому случаю планирования перемещения робота-точки. Другое применение суммы Минковского — задача упрощения формы многоугольников (см. *разд. 20.12*). Здесь мы делаем границу объекта толще, чтобы создать вокруг него канал, а потом выясняем упрощенную форму объекта, для чего находим путь внутри этого канала, состоящий из минимального количества отрезков. Наконец, сумма Минковского для объекта, имеющего очень неровную границу, и небольшого круга сглаживает его границу, устраняя небольшие впадины и выступы.

Определение суммы Минковского выглядит следующим образом (при этом предполагается, что многоугольники $A$ и $B$ находятся в некоторой системе координат):

$$A + B = \{x + y \mid x \in A, y \in B\},$$

где $x + y$ — сумма соответствующих векторов. В терминах операции переноса сумма Минковского — это объединение всех переносов многоугольника $A$ на расстояние, определяемое точкой внутри многоугольника $B$. При вычислении суммы Минковского возникают следующие вопросы.

- *Входные объекты представляют собой растровые изображения или многоугольники?*

  Если $A$ и $B$ являются растровыми изображениями, то определение суммы Минковского подсказывает простой алгоритм для ее вычисления. Инициализируем достаточно большую матрицу пикселов, вычислив сумму Минковского ограничивающих прямоугольников для многоугольников $A$ и $B$. Для каждой пары точек из $A$ и $B$ складываем их координаты и уменьшаем яркость соответствующего пиксела. Эти алгоритмы становятся более сложными, если требуется явное представление суммы Минковского в виде многоугольника.

- *Требуется ли увеличить размер объекта на заданное значение?*

  При типичной операции увеличения объекта размер модели $M$ увеличивается на определенное значение допуска $t$, называемое *смещением* (offsetting). Как показано на рис. 20.16, результат достигается путем вычисления суммы Минковского для модели $M$ и круга радиусом $t$. Базовые алгоритмы продолжают работать, хотя полученный объект не является многоугольником, поскольку теперь его граница состоит из дуг и отрезков.

- *Являются ли входные объекты выпуклыми?*

  Сложность вычисления суммы Минковского во многом зависит от формы многоугольников. Когда оба многоугольника выпуклые, то сумму Минковского можно вычислить за время $O(n + m)$, проведя один многоугольник по контурам другого. Если один из многоугольников не является выпуклым, то *размер* результирующего объекта (количество вершин или ребер) может достичь значения $\Theta(nm)$. А если же невыпуклыми являются оба многоугольника, то и значения $\Theta(n^2m^2)$. Суммы Мин-

ковского для невыпуклых многоугольников часто имеют эстетически непривлекательный вид, поскольку отверстия в них возникают или исчезают самым неожиданным образом.

Простейший подход к вычислению сумм Минковского основан на триангуляции и объединении. Сначала выполняем триангуляцию каждого многоугольника, а затем вычисляем сумму Минковского для каждого треугольника из $A$ с каждым треугольником из $B$. Сумма двух треугольников является легко вычисляемым частным случаем суммы выпуклых многоугольников, рассматриваемым далее. Объединением этих $O(nm)$ выпуклых многоугольников будет сумма $A + B$. Алгоритмы поиска объединения многоугольников основаны на методе заметающей прямой (см. *разд. 20.8*).

Вычислить сумму Минковского для двух выпуклых многоугольников легче, чем для общего случая, поскольку эта сумма всегда будет выпуклой. Когда многоугольники выпуклые, проще перемещать многоугольник $A$ вдоль границы многоугольника $B$ и вычислять сумму для каждого ребра. Подход, в котором каждый многоугольник разбивается на небольшое количество выпуклых фрагментов (см. *разд. 20.11*), а потом вычисляется сумма Минковского для каждой пары фрагментов, обычно эффективнее, чем обработка двух многоугольников, полностью подвергнутых триангуляции.

### Реализации

Пакет библиотеки CGAL (**www.cgal.org**) предоставляет эффективные процедуры расчета суммы Минковского для двух произвольных многоугольников, а также вычисления точных и приблизительных смещений.

Реализация алгоритма вычисления суммы Минковского для двух выпуклых многогранников в трех измерениях описана в работе [FH06] и доступна по адресу **https://www.cs.tau.ac.il/~efif/CD/**.

#### *Примечания*

Обсуждение алгоритмов вычисления суммы Минковского можно найти в книгах [dBvKOS08] и [O'R01]. Самые быстрые алгоритмы вычисления суммы Минковского представлены в работах [KOS91] и [Sha87].

Практическая эффективность вычисления суммы Минковского в общем случае зависит от того, каким образом многоугольники разбиты на выпуклые фрагменты. Разбиение многоугольников на минимальное количество выпуклых фрагментов не обязательно будет оптимальным решением. Всестороннее обсуждение методов разбиения многоугольников для вычисления суммы Минковского приведено в работе [AFH02]. В работе [BFH+18] демонстрируется, как ускорить вычисление суммы Минковского для многоугольников с отверстиями, пользуясь тем, что достаточно небольшие отверстия не влияют на форму результирующего многоугольника.

Комбинаторная сложность суммы Минковского для двух выпуклых многогранников в трех измерениях полностью определена в работе [FHW07]. Реализация алгоритма вычисления суммы Минковского для таких многогранников описана в работе [FH06].

В работе [KS90] представлен эффективный алгоритм, основанный на вычислении сумм Минковского для планирования перемещений роботов многоугольной формы.

### Родственные задачи

Преобразования к срединной оси (см. *разд. 20.10*), планирование перемещений (см. *разд. 20.14*), упрощение многоугольников (см. *разд. 20.12*).

# Множества и строки

Как множества, так и строки представляют собой коллекции объектов, и разница между ними состоит в том, имеет ли значение порядок элементов коллекции. *Множества* — это группы символов, порядок которых не имеет значения, а *строки* определяются как последовательность символов.

Тот факт, что элементы в строках упорядочены, позволяет решать задачи со строками намного эффективнее, чем задачи с множествами, благодаря возможности использовать методы динамического программирования и развитые структуры данных типа суффиксных деревьев. Причиной повышения интереса, проявляемого к алгоритмам обработки строк, и важности этих алгоритмов стали нужды биоинформатики, средства социального общения и другие приложения обработки текста. Среди хороших книг по алгоритмам для обработки строк можно назвать следующие:

- [Gus97]) — пожалуй, самое лучшее введение в обработку строк. Эта книга содержит подробное обсуждение суффиксных деревьев, а также современные формулировки классических алгоритмов для точного сравнения строк;
- [CHL07] — подробное описание алгоритмов обработки строк, автор которого является признанным лидером в этой области (перевод с французского на английский);
- [NR07] — краткое, но имеющее практическую ценность обсуждение алгоритмов поиска по образцу, ориентированное на конкретные реализации. Особое внимание уделяется подходам, в которых применяется параллелизм на уровне битов;
- [CR03] — обзор некоторых специальных тем, имеющих отношение к алгоритмам обработки строк, с акцентом на теорию.

Ежегодная конференция CPM (Combinatorial Pattern Matching — комбинаторное сравнение строк) считается основным форумом, посвященным практическим и теоретическим аспектам алгоритмов обработки строк.

## 21.1. Поиск покрытия множества

**Вход.** Коллекция подмножеств $S = \{S_1, ..., S_m)$ универсального множества $U = \{1, ..., n\}$.

**Задача.** Найти наименьшую коллекцию $T$ подмножеств множества $S$, объединение которых равно универсальному множеству, т. е.

$$\bigcup_{i=1}^{|T|} T_i = U.$$

### Обсуждение

Задача о покрытии множества (рис. 21.1) возникает, когда мы стремимся экономно приобрести товары, разложенные по наборам. Мы хотим получить как минимум по

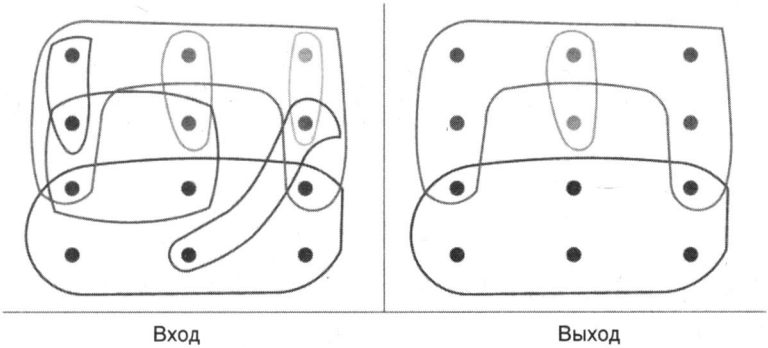



**Рис. 21.1.** Покрытие множества

одному товару каждого вида, покупая при этом как можно меньшее количество наборов. Задача поиска *хоть какого-нибудь* покрытия множества не представляет сложности, поскольку можно купить все предлагаемые наборы товаров. Но поиск наименьшего покрытия множества позволяет нам минимизировать наши расходы. Задача о покрытии множества позволила упростить формулировку задачи оптимизации выбора лотерейных билетов, обсуждавшуюся в *разд. 1.8*. В той задаче нам требовалось купить наименьшее количество лотерейных билетов, покрывающее все комбинации заданного набора.

Другим интересным применением задачи о покрытии множества представляется задача оптимизации булевой логики. Рассмотрим, например, булеву функцию с $k$ переменными, возвращающую 0 или 1 для каждого из возможных $2^k$ входных векторов. Нам нужно найти простейшую логическую схему, которая реализует эту функцию. Один из подходов — определить на этих переменных и их дополнениях булеву формулу в *дизъюнктивной нормальной форме* — например, $x_1\overline{x}_2 + \overline{x}_1\overline{x}_2$. Для каждого входного вектора мы могли бы построить один член И, а потом выполнить операцию ИЛИ над всеми этими членами И, однако мы существенно сэкономим, если вынесем за скобки общие подмножества переменных. Имея набор выполнимых членов И, каждый из которых покрывает какое-либо подмножество нужных нам векторов, мы можем объединить операцией ИЛИ наименьшее количество членов, которые реализуют заданную функцию. Это как раз и есть задача о покрытии множества.

Существует несколько разновидностей задачи о покрытии множества, и для их распознавания нужно ответить на следующие вопросы.

- *Разрешено ли повторное включение элементов в покрытие?*

  Если любой элемент может входить только в одно подмножество, то задача о покрытии множества превращается в задачу *укладки множества*, которая рассматривается в *разд. 21.2*. Так что если есть возможность включать один элемент в несколько подмножеств, то следует воспользоваться ею, поскольку в результате обычно удается получить меньшее покрытие.

- *В задаче рассматривается множество ребер или множество вершин графа?*

  Задача о покрытии множества имеет общий характер и включает в себя несколько полезных задач на графах в качестве частных случаев. Допустим, что нужно найти

наименьшее множество ребер графа, которое затронет каждую вершину только один раз. Это на самом деле задача о *совершенном паросочетании* в графе (см. *разд. 18.6*). А теперь допустим, что нужно найти наименьшее множество вершин графа, которое задействует каждое ребро по крайней мере один раз. Эта задача является задачей о вершинном покрытии, и эвристические методы для ее решения рассматриваются в *разд. 19.3*.

Здесь полезно увидеть, как моделируется вершинное покрытие в виде экземпляра покрытия множества. Пусть универсальное множество $U$ соответствует множеству ребер $\{e_1, ..., e_m\}$. Создаем $n$ подмножеств, где подмножество $S_i$ состоит из ребер, инцидентных вершине $v_i$. Хотя задача о вершинном покрытии является частным случаем задачи о покрытии множества, имеет смысл воспользоваться более эффективными эвристическими методами решения частной задачи о вершинном покрытии.

- *Каждое подмножество содержит только два элемента?*

  Если в любом подмножестве содержится самое большее два элемента, считайте, что вам повезло. Для этого частного случая можно получить оптимальное решение, поскольку он сводится к поиску максимального паросочетания графа. К сожалению, как только количество элементов во всех подмножествах возрастает до трех, задача становится NP-полной.

- *Нужно найти множества, содержащие элементы, или элементы, содержащиеся во множествах?*

  В *задаче о минимальном множестве представителей* (hitting set) нужно найти множество элементов, которые совместно представляют каждое подмножество из имеющейся коллекции. Оптимальное решение экземпляра задачи о минимальном множестве представителей получается при выборе элементов 1 и 3 или 2 и 3 (рис. 21.2, *слева*). Эту задачу можно преобразовать в экземпляр двойственной задачи о покрытии множества, оптимальным решением которой будет выбор подмножеств 1 и 3 или подмножеств 2 и 4 (рис. 21.2, *справа*).

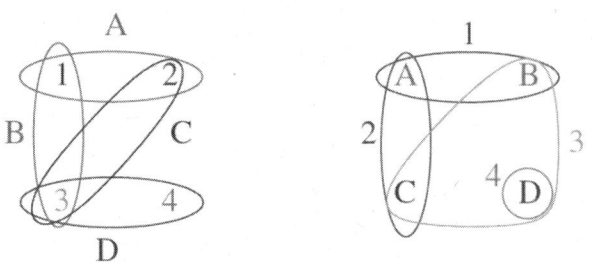



**Рис. 21.2.** Примеры оптимальных решений: экземпляра задачи о минимальном множестве представителей (*слева*) и экземпляра двойственной задачи о покрытии множества (*справа*)

Входной экземпляр задачи о минимальном множестве представителей идентичен входному экземпляру задачи о покрытии множества, но здесь требуется найти такое минимальное подмножество элементов $T \subset U$, чтобы каждое подмножество $S_i$ содержало по крайней мере один элемент подмножества $T$. Таким образом, $S_i \cap T \neq 0$

для всех $1 \le i \le m$. Допустим, что мы хотим создать небольшой парламент, в который входил бы как минимум один представитель от каждой демографической группы. Люди одновременно обладают несколькими демографическими характеристиками, и поэтому могут одновременно принадлежать к нескольким группам. Например, я мужчина, еврейской этничности, левша и бэби-бумер[1]. Если каждая группа определяется как определенное подмножество всего населения, то решение задачи о минимальном множестве представителей и будет решением задачи создания политкорректного парламента.

Задача о минимальном множестве представителей двойственна задаче о покрытии множества. Заменим каждый элемент множества $U$ множеством имен подмножеств, содержащих его. Теперь множества $S$ и $U$ поменялись ролями, поскольку мы ищем множество подмножеств множества $U$, чтобы покрыть все элементы в множестве $S$. Мы получили задачу о покрытии множества, поэтому можем использовать любую программу для ее решения, чтобы решить задачу о минимальном множестве представителей (см. рис. 21.2).

Задача о покрытии множества по меньшей мере так же сложна, как и задача о вершинном покрытии, поэтому она тоже является NP-полной. На самом же деле она еще сложнее. Решение, выдаваемое аппроксимирующими алгоритмами для задачи о вершинном покрытии, хуже оптимального не более чем в два раза, а для задачи о покрытии множества самое лучшее решение хуже оптимального в $\Theta(\lg n)$ раз.

Самый естественный и эффективный подход к решению задачи о покрытии множества — использование «жадного» эвристического алгоритма. Для начала выбираем самое мощное подмножество для покрытия, после чего удаляем все его элементы из универсального множества. Добавляем подмножество, содержащее наибольшее количество неохваченных элементов, и повторяем это действие, пока все элементы не будут покрыты. Количество подмножеств покрытия множества, выдаваемых таким эвристическим алгоритмом, никогда не превысит оптимальное более чем в $\ln n$ раз, причем на практике этот коэффициент намного меньше.

Простейшая реализация «жадного» эвристического алгоритма просматривает на каждом шаге весь входной экземпляр из $m$ подмножеств. Впрочем, используя такие структуры данных, как связные списки и очереди с приоритетами ограниченной высоты (см. *разд. 15.2*), можно реализовать «жадный» эвристический алгоритм с временем исполнения $O(S)$, где $S = \bigcup_{i=1}^{m} |S_i|$ — размер входного экземпляра.

Полезно проверить существование элементов, содержащихся лишь в небольшом количестве подмножеств — в идеале только в одном. При наличии таких элементов следует выбрать наибольшее подмножество, содержащее этот элемент, в самом начале работы алгоритма. В конечном счете нам все равно придется выбрать такое подмножество, но оно содержит другие элементы, покрытие которых потребует дополнительных затрат, если выбрать это подмножество не с самого начала.

---

[1] Бэби-бумеры (поколение бэби-бума) — послевоенное поколение, люди, рожденные в 1946–1964 годах. Это теперешние бабушки и дедушки, выросшие до появления Интернета. — *Прим. ред.*

Применение методов имитации отжига поверх предложенных эвристических подходов, скорее всего, даст несколько лучшие результаты. Чтобы гарантировать оптимальное решение, можно использовать поиск с возвратом, но выгода от этого не часто оправдывает дополнительные расходы.

Еще один, часто более мощный подход основан на переформулировке задачи о покрытии множества в терминах целочисленного линейного программирования. Пусть целая переменная $s_i$, принимающая два значения: 0 и 1, обозначает, выбрано ли подмножество $S_i$ для заданного покрытия. Для каждого элемента $x$ из универсального множества определяется ограничение

$$\sum_{x \in S_i} s_i \geq 1$$

на основе всех содержащих его подмножеств $S_i$. Таким образом гарантируется, что элемент $x$ будет покрыт хотя бы одним выбранным подмножеством. Минимальное покрытие множества удовлетворяет всем этим ограничивающим условиям и одновременно минимизирует $\sum_i s_i$. Эту задачу целочисленного программирования можно с легкостью обобщить до задачи о взвешенном покрытии множества (если допустить разную стоимость разных подмножеств). Ослабив эту задачу до задачи линейного программирования (т. е. позволив каждой переменной $s_i$ находиться в диапазоне $0 \leq s_i \leq 1$, не ограничивая ее только двумя значениями 0 или 1), можно получить эффективный эвристический алгоритм, основанный на методах округления.

### Реализации

Как «жадный» эвристический подход, так и подход с использованием целочисленного линейного программирования являются настолько простыми в своей области, что их можно реализовать с чистого листа.

Реализация на языке Pascal алгоритма исчерпывающего перебора для решения задачи укладки множества, а также для задачи о покрытии множества дается в книге [SDK83]. Подробности см. в *разд. 22.1.9*.

Пакет SYMPHONY содержит процедуру для решения задачи разбиения множества методами смешанного линейного программирования. Загрузить ее можно с веб-сайта **https://github.com/coin-or/SYMPHONY**.

#### *Примечания*

В работе [BP76] представлен классический обзор методов решения задачи о покрытии множества, а более свежий обзор аппроксимирующих методов с анализом их сложности приведен в работе [Pas97]. Результаты исследований эвристических методов целочисленного программирования и точные алгоритмы решения задачи о покрытии множества изложены в работах [CFT99] и [CFT00]. Убедительное обсуждение алгоритмов и правил сведения для задачи о покрытии множества содержится в книге [SDK83].

Среди хороших работ, посвященных «жадным» эвристическим алгоритмам решения задачи о покрытии множества, можно назвать книги [CLRS09] и [Hoc96]. Пример, демонстрирующий, что решение задачи о покрытии множества, выдаваемое «жадным» эвристическим алгоритмом, может быть в $\lg n$ хуже оптимального, представлен в работах [Joh74] и [PS98]. Впрочем, такой результат не является следствием дефекта в алгоритме. Доказана

сложность получения приблизительного решения задачи о покрытии множества с коэффициентом, лучшим чем $(1 - o(1)) \ln n$ (см. [Fei98]). Для задач ограниченных покрытий множеств, возникающих из геометрических экземпляров (например, вычисления наименьшего количества точек для получения множества представителей для заданного множества кругов), возможны лучшие результаты (см. [AP14] и [MR10]).

В Томе 4A Дональда Кнута ([Knu11]) приведено увлекательное обсуждение оптимизации булевой логики, похожей на задачу покрытия множества.

### Родственные задачи

Паросочетание (см. *разд. 18.6*), вершинное покрытие (см. *разд. 19.3*), укладка множества (см. *разд. 21.2*).

## 21.2. Задача укладки множества

**Вход.** Множество подмножеств $S = \{S_1, ..., S_m\}$ универсального множества $U = \{1, ..., n\}$.

**Задача.** Выбрать небольшую коллекцию *взаимно непересекающихся* подмножеств из множества $S$, объединением которых является универсальное множество (рис. 21.3).

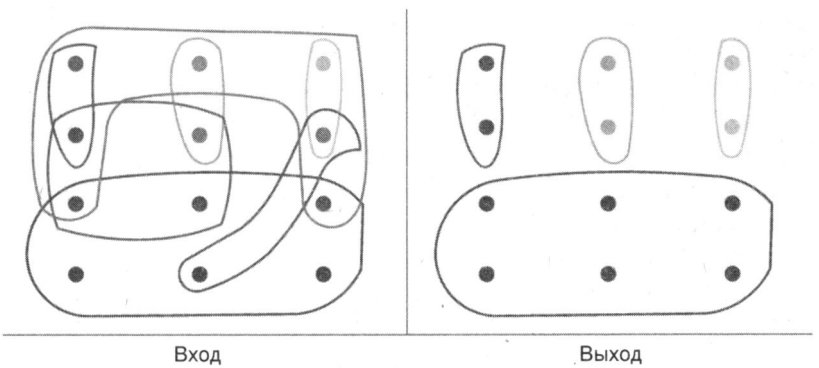



**Рис. 21.3.** Пример укладки множества

### Обсуждение

Задачи укладки множества возникают в приложениях, имеющих строгие ограничивающие условия на разрешенное разбиение. Главной особенностью задач укладки множества (в отличие от задач покрытия множества) является условие, согласно которому ни один элемент не может быть покрыт больше чем одним выбранным подмножеством.

Эта задача в определенной степени аналогична задаче поиска независимого множества в графах (см. *разд. 19.2*), где требуется найти наибольшее подмножество вершин графа $G$, в котором каждое ребро инцидентно не более чем одной из выбранных вершин. Смоделируем задачу поиска подмножества вершин в виде задачи укладки множества. Пусть универсальное множество состоит из всех ребер графа $G$, а подмножество $S_i$ — из всех ребер, инцидентных вершине $v_i$. Дополнительно определим одноэлементное множество для каждого ребра. Любая укладка множества определяет множество вер-

шин, не имеющих общих ребер, — иными словами, независимое множество. Одноэлементные множества используются для покрытия ребер, не охваченных выбранными вершинами.

Еще одно применение укладки множества — комплектование экипажей самолетов. На каждый самолет авиакомпании нужно назначить экипаж, состоящий из двух пилотов и штурмана. К составу экипажа предъявляются определенные требования — такие как умение управлять самолетом именно этого типа, психологическая совместимость, учитывается также и рабочее расписание. Для всех возможных комбинаций экипажей и самолетов, где каждая представлена подмножеством элементов, нужно найти такой вариант, при котором каждый самолет и каждый член экипажа присутствуют только в одной комбинации. Ведь один и тот же человек не может одновременно находиться на двух разных самолетах, а каждому самолету нужен полноценный экипаж. То есть нам надо получить идеальную укладку с учетом имеющихся ограничений подмножества.

Задача укладки множества включает в себя несколько типов NP-полных задач на множествах. Для их распознавания нужно ответить на следующие вопросы.

- *Обязательно ли, чтобы каждый элемент входил только в одно выбранное подмножество?*

  В задаче о точном покрытии требуется найти коллекцию подмножеств, в которой каждый элемент покрыт только один раз. Рассмотренная только что задача комплектования экипажей самолетов близка к задаче точного покрытия, поскольку в ней должны быть задействованы каждый самолет и каждый экипаж.

  К сожалению, задача поиска точного покрытия ставит нас в такую же ситуацию, как и задача поиска гамильтонова цикла в графах. Если *действительно* нужно покрыть все элементы только по одному разу, а эта задача является NP-полной, то единственным способом ее решения является поиск с экспоненциальным временем исполнения. Стоимость такого подхода окажется очень высокой, если только вам не повезет и вы не натолкнетесь на решение в начале поиска.

- *Есть ли у каждого элемента свое одноэлементное множество?*

  Ситуация намного улучшается, если нам достаточно частичного решения — например, такого, при котором каждый элемент универсального множества $U$ определяется как одноэлементное подмножество множества $S$. Тогда мы можем расширить любую укладку множества до точного покрытия, покрыв не попавшие в укладку элементы $U$ одноэлементными множествами. После этого наша задача сводится к поиску укладки множества минимальной мощности, решение которой можно получить с помощью эвристических методов.

- *Какова плата за двойное покрытие элементов?*

  В задаче о покрытии множества (см. *разд. 21.1*) включение одного элемента в несколько выбранных подмножеств не влечет за собой никаких отрицательных последствий. Однако в задаче точного покрытия многократные вхождения запрещены. Впрочем, во многих приложениях запреты не столь строгие. В одном из подходов к решению таких задач можно увеличить стоимость выбора подмножеств, содержащих элементы, входящие в уже выбранные подмножества.

Самым лучшим подходом к решению задачи укладки множества является использование «жадных» эвристических алгоритмов, подобных тем, что применяются для решения задачи о покрытии множества (см. *разд. 21.1*). Если нам требуется найти укладку, содержащую наименьшее количество подмножеств, мы выбираем наибольшее подмножество, удаляем все конфликтующие с ним подмножества из множества $S$ и повторяем процесс. Как обычно, если мы дополним этот подход каким-либо методом исчерпывающего поиска или рандомизации (например, методом имитации отжига), то, вероятно, получим более удачные укладки за счет увеличения времени работы.

Еще один, более мощный подход основан на переформулировке задачи о покрытии множества в терминах целочисленного программирования, подобно тому, как мы делали это в задаче о покрытии множества. Пусть целая переменная $s_i$, принимающая два значения: 0 и 1, обозначает, выбрано ли подмножество $S_i$ для того или иного покрытия. Для каждого элемента $x$ из универсального множества $U$ определяется ограничение

$$\sum_{x \in S_i} s_i = 1$$

на основе всех содержащих его подмножеств $S_i$. Таким образом гарантируется, что элемент $x$ будет покрыт *только* одним выбранным подмножеством. Минимизация или максимизация $\sum_i s_i$ при соблюдении этих ограничивающих условий позволяет нам регулировать количество подмножеств в покрытии.

### Реализации

Поскольку задача о покрытии множества более популярна и легче решается, чем задача укладки множества, для ее решения проще найти подходящую реализацию. Рассматриваемые в *разд. 21.1* реализации без труда поддаются модификации, что позволяет соблюдать конкретные ограничивающие условия задачи упаковки множества.

Реализация на языке Pascal алгоритма исчерпывающего перебора для решения задачи упаковки множества, а также для задачи о покрытии множества приведена в книге [SDK83]. Информацию по загрузке этих программ можно найти в *разд. 22.1.9*.

Пакет SYMPHONY содержит процедуру для решения задачи разбиения множества методами смешанного линейного программирования. Загрузить ее можно с веб-сайта **https://github.com/coin-or/SYMPHONY**.

#### *Примечания*

Среди обзорных статей, посвященных задаче укладки множества, можно рекомендовать следующие [BP76], [HP09] и [Pas97]. В работе [SW13] представлен эвристический алгоритм локального поиска для решения задачи упаковки множества. Также изучались фиксированно-параметрически разрешимая и онлайновая[2] версии алгоритмов для решения задачи упаковки множества — см. [FKN+08] и [EHM+12] соответственно. Стратегии предложения цены в аукционах, в которых товары скомплектованы в наборы, обычно сводятся к решению задач укладки множества, как описывается в работе [dVV03].

[2] В указанном случае это означает алгоритм, обрабатывающий по одному элементу ввода за раз, не зная о значениях будущих элементов ввода. — *Прим. пер.*

Ослабленные ограничивающие условия для задач целочисленного программирования, соответствующих задаче укладки множества, представлены в работе [BW00]. Убедительное обсуждение алгоритмов и правил сведения для задачи о покрытии множества представлено в книге [SDK83]. В ней же вы найдете описанное ранее приложение комплектования экипажей самолетов.

### Родственные задачи

Независимое множество (см. *разд. 19.2*), вершинное покрытие (см. *разд. 19.3*).

## 21.3. Сравнение строк

**Вход.** Текстовая строка $t$ из $n$ символов, строка-образец $p$ длиной в $m$ символов.

**Задача.** Найти в текстовой строке $t$ первое или все вхождения строки-образца $p$ (рис. ЦВ-21.4).

### Обсуждение

Задача сравнения строк возникает почти во всех приложениях обработки текста. Любой текстовый редактор имеет механизм поиска произвольной строки в текущем документе. Языки программирования, такие как Python, обладают широкими возможностями поиска подстрок благодаря наличию встроенных примитивов сравнения строк, и это позволяет писать на них программы, способные фильтровать и модифицировать текст. Наконец, программы проверки правописания ищут в своем словаре каждое слово проверяемого текста и помечают слова, отсутствующие в словаре.

Несмотря на свою фундаментальность, алгоритмическая задача сравнения строк продолжает активно исследоваться. При выборе правильного алгоритма сравнения строк для конкретного приложения возникает несколько вопросов.

- *Какова длина образцов для поиска?*

  При недлинных образцах и нечастых запросах на поиск достаточно использовать простой алгоритм поиска со временем исполнения $O(mn)$. Для всех возможных начальных позиций $1 \le i \le n - m + 1$ в строке поиска выполняется проверка подстроки длиной $m$ символов на идентичность со строкой-образцом. Реализация этого алгоритма на языке C приводится в *разд. 2.5.3*.

  При очень коротких образцах ($m \le 10$) нет смысла пытаться улучшить этот простой алгоритм в надежде на повышение производительности. Кроме того, для типичных строк ожидаемое время исполнения будет намного лучше, чем $O(mn)$, поскольку мы продвигаем образец дальше, как только обнаруживаем его несовпадение с подстрокой текста. Более того, этот простой алгоритм обычно исполняется за линейное время. Впрочем, наихудший случай вполне вероятен, — возьмем, например, образец $p = a^m$ и текст $t = (a^{m-1}b)^{n/m}$.

- *Как поступать с длинными текстами и образцами?*

  В наихудшем случае поиск строк может осуществляться за линейное время. Обратите внимание, что после обнаружения несовпадающих символов нет необходимости возобновлять поиск с начала, поскольку префикс образца и текст одинаковы вплоть до точки несовпадения. Имея частичное совпадение, заканчивающееся в по-

зиции $i$, мы переходим на первый символ в образце или тексте, который может предоставить новую информацию о тексте в позиции $i + 1$. Алгоритм Кнута — Морриса — Пратта выполняет предварительную обработку образца для эффективного создания такой таблицы переходов. Подробности правильной его реализации весьма сложны, но конечный алгоритм имеет довольно небольшой размер.

- *Велика ли вероятность найти совпадение с образцом?*

  Алгоритм Бойера — Мура сравнивает образец с текстом справа налево, что позволяет избежать просмотра больших фрагментов текста при отсутствии совпадения. Допустим, что для образца поиска используется строка *abracadabra*, а в одиннадцатой позиции в тексте находится буква *x*. Этот образец никоим образом не может совпасть с первыми одиннадцатью символами текста, поэтому следующей точкой проверки в тексте является двадцать второй символ. Если нам повезет, то проверить придется всего лишь $n/m$ символов. В случае несовпадения алгоритм Бойера — Мура использует два набора таблиц переходов: один построен на основе текущих совпадений, а второй — на символе, вызвавшем несовпадение.

  Хотя этот алгоритм сложнее, чем алгоритм Кнута — Морриса — Пратта, он оправдывает себя на практике для строк-образцов длиной свыше десяти символов, при условии отсутствия частых вхождений образца в текст поиска. Время исполнения алгоритма Бойера — Мура в наихудшем случае равно $O(n + rm)$, где $r$ — количество вхождений образца $p$ в текст $t$.

- *Будет ли выполняться повторный поиск в одном и том же тексте?*

  Допустим, что вы создаете программу для многократных запросов на поиск в некоторой текстовой базе. Так как текст поиска не изменяется, имеет смысл создать структуру данных, позволяющую ускорить выполнение запросов. Подходящими структурами данных для этой цели могут стать суффиксные деревья и суффиксные массивы, рассматриваемые в *разд. 15.3*.

- *Планируется ли поиск одного и того же образца в разных текстах?*

  Допустим, что вы создаете программу, отфильтровывающую из текста нецензурные выражения. В этом случае набор образцов остается постоянным, а текст, в котором производится поиск, будет меняться. Нам может потребоваться найти все вхождения в текст любого из $k$ разных образцов, при этом значение $k$ может быть довольно большим.

  Линейное время для поиска каждого образца означает алгоритм с общим временем исполнения $O(k(m + n))$. Но для больших значений $k$ существует лучшее решение, создающее один конечный автомат, который распознает эти образцы и возвращается в соответствующее начальное состояние при любом несовпадении символов. Алгоритм Ахо — Корасика создает такой автомат за линейное время. Оптимизируя автоматы распознавания образов (см. *разд. 21.7*), можно добиться экономии памяти. Этот подход был использован в первоначальной версии программы fgrep.

  Иногда несколько образцов можно указать не в виде списка строк, а сжато, в форме регулярного выражения. Например, регулярное выражение $a(a + b + c)^*a$ соответствует любой строке алфавита ($a$, $b$, $c$), которая начинается и заканчивается на букву $a$. Самый лучший способ проверки, распознается ли входная строка тем или иным

регулярным выражением $R$, заключается в создании конечного автомата, эквивалентного $R$, с последующей эмуляцией этого автомата на заданной строке. Подробности создания автомата на основе регулярных выражений приводятся в *разд. 21.7*.

Когда вместо регулярных выражений для указания образцов используются контекстно-свободные грамматические выражения, задача превращается в задачу синтаксического разбора (см. *разд. 10.8*).

- *Как поступить, если текст или образец содержит орфографические ошибки?*

  Рассматриваемые здесь алгоритмы пригодны только для точного сравнения строк. Если необходим допуск на орфографические ошибки, то задача превращается в задачу *нечеткого сравнения строк* (см. *разд. 21.4*).

### Реализации

Программы на языке C коллекции strmat реализуют алгоритмы для точного сравнения образцов, включая реализацию нескольких вариантов алгоритмов Кнута — Морриса — Пратта и Бойера — Мура. Самым лучшим справочным материалом по этим алгоритмам является книга [Gus97]. Загрузить пакет можно с веб-сайта **https://www.cs.ucdavis.edu/~gusfield/strmat.html**.

Свободно доступны несколько версий программы сравнения регулярных выражений grep. Вариант GNU программы grep можно найти по адресу **http://directory.fsf.org/project/grep/**. Эта версия GNU grep использует гибрид быстрого детерминистского алгоритма отложенных состояний с алгоритмом Бойера — Мура для поиска строк фиксированной длины.

Библиотека Boost содержит реализации на языке C++ алгоритмов обработки строк, включая поиск. Подробная информация об этих реализациях доступна по адресу **https://www.boost.org/doc/libs/1_77_0/doc/html/string_algo.html**.

#### Примечания

Все книги по алгоритмам обработки строк, включая следующие [CHL07], [NR07] и [Gus97], содержат исчерпывающее обсуждение точного сравнения строк. Хорошие описания алгоритмов Бойера — Мура (см. [BM77]) и Кнута — Морриса — Пратта (см. [KMP77]) представлены в книгах [BvG99], [CLRS09] и [Man89]. В истории создания алгоритмов сравнения строк были и неудачи — некоторые из опубликованных работ содержали ошибки (подробности см. в книге [Gus97]).

В книге [Aho90] приводится хороший обзор алгоритмов поиска образцов в строках — в частности, алгоритмов поиска с помощью регулярных выражений. Алгоритм Ахо — Корасика описывается в работе [AC75].

Эксперименты по сравнению алгоритмов обработки строк представлены в книгах [DB86], [Hor80], [Lec95], [YLDF16] и [dVS82]. Качество работы каждого конкретного алгоритма зависит от свойств строк и размера алфавита. Для работы с длинными образцами и большими текстами я рекомендую использовать самые лучшие реализации алгоритма Бойера — Мура, которые вы сможете найти. В статье [LLCC12] рассматриваются алгоритмы для сравнения строк, исполняющиеся на графических процессорах.

В алгоритме Рабина — Карпа (см. [KR87]) для сравнения строк используется хеш-функция, что позволяет получить линейное ожидаемое время исполнения. Но время исполнения этого алгоритма в наихудшем случае остается квадратичным, а его производи-

тельность на практике оказывается несколько хуже, чем у описанных ранее методов сравнения символов. Этот алгоритм обсуждается в *разд. 6.7*.

### Родственные задачи

Суффиксные деревья (см. *разд. 15.3*), нечеткое сравнение строк (см. *разд. 21.4*).

## 21.4. Нечеткое сравнение строк

**Вход.** Текстовая строка $t$ и строка-образец $p$.

**Задача.** Найти самый дешевый способ преобразования строки $t$ в строку $p$, в котором используются вставки, удаления и замены символов (рис. 21.5).

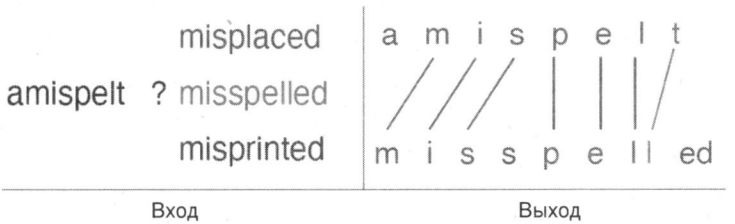



**Рис. 21.5.** Нечеткое сравнение строк

### Обсуждение

Задача нечеткого сравнения строк весьма важна по той причине, что мы живем в мире, где нет четких границ. Программы проверки правописания должны находить наиболее подходящее слово для любой строки символов, отсутствующей в словаре (т. е. в базе данных образцов). Возможность эффективного поиска сходных фрагментов в больших базах данных нуклеотидных последовательностей ДНК коренным образом изменила направление исследований в молекулярной биологии. Допустим, вы исследовали некоторый ген человека, и оказалось, что он очень похож на какой-либо ген крысы. Скорее всего, этот ген выполняет одинаковую функцию как в организме человека, так и в организме крысы, а разница между этими его двумя версиями является результатом эволюционных мутаций.

Мне однажды пришлось столкнуться с задачей нечеткого сравнения строк при оценке качества работы системы оптического распознавания текста. В ней нужно было сравнить ответы, выдаваемые системой для тестового документа, с правильными результатами. Чтобы улучшить систему, нам требовалось идентифицировать ошибочно распознаваемые буквы. Аналогичный принцип используется в программах сравнения файлов, которые выявляют строки, различающиеся в двух разных версиях файла.

Когда изменения недопустимы, задача сводится к точному сравнению строк, которая рассматривается в *разд. 21.3*. Здесь же мы будем рассматривать только задачу сравнения строк с ошибками.

Динамическое программирование предоставляет нам базовый подход к нечеткому сравнению строк. Пусть $D[i, j]$ обозначает стоимость редактирования первых $i$ символов строки образца $p$ в первые $j$ символов текста $t$. Мы должны выполнить *какое-либо*

действие с концевыми символами $p_i$ и $t_j$. Нам доступны только следующие возможности: зафиксировать совпадение или заменить один символ другим, удалить $p_i$ или, наконец, вставить символ, совпадающий с $t_j$. Таким образом, значение $D[i, j]$ является минимальным из следующих трех значений стоимости:

- если $p_i = t_j$, то $D[i-1, j-1]$, иначе $D[i-1, j-1]$ + стоимость замены;
- $D[i-1, j]$ + стоимость удаления $p_i$;
- $D[i, j-1]$ + стоимость удаления $t_j$.

Общая реализация этого алгоритма на языке C и более подробное его обсуждение приводятся в *разд. 10.2*. Прежде чем использовать это рекуррентное соотношение, ответьте на несколько вопросов.

- *Образец должен совпасть со всем текстом или только с подстрокой?*

  Разница между алгоритмами сравнения строк и алгоритмами сравнения подстрок определяется граничными условиями этого рекуррентного соотношения. Допустим, что нам нужно сравнить весь образец со всем текстом. Тогда стоимость $D[i, 0]$ должна равняться стоимости удаления первых $i$ символов образца, поэтому $D[i, 0] = i$. Подобным образом $D[0, j] = j$.

  Но теперь допустим, что образец может встретиться в любом месте текста. Корректным значением стоимости $D[0, j]$ становится 0, поскольку в этом случае не должно быть штрафа за то, что выравнивание начинается в $j$-й позиции текста. Стоимость $D[i, 0]$ по-прежнему остается равной $i$, т. к. единственным способом получения совпадения первых $i$ символов образца с пустой строкой является удаление всех этих символов. Стоимость самого лучшего совпадения образца с подстрокой будет $\min_{k=1}^{n} D[m, k]$.

- *Как определить стоимость операций замены, вставки и удаления?*

  Алгоритм нахождения расстояния редактирования может использовать разную стоимость вставки, удаления и замены символов. Конкретное значение стоимости каждой операции зависит от того, что вы намереваетесь делать с выровненными текстами.

  Вариант по умолчанию — назначение одинаковой стоимости для всех операций вставки, удаления и замены. Более высокая стоимость замены, чем суммарная стоимость вставки и удаления, гарантирует, что замена никогда не будет выполнена, поскольку простое редактирование вне строки обойдется дешевле. Выполнение только операции вставки и удаления сводит исходную задачу к задаче поиска максимальной общей подстроки, рассматриваемой в *разд. 21.8*. Часто бывает полезно слегка изменить стоимость расстояний редактирования и исследовать полученные результаты, повторяя процесс до тех пор, пока не будут определены наилучшие параметры для вашей задачи.

- *Как узнать, какие операции фактически привели к выравниванию строк?*

  Ранее рассмотренное рекуррентное соотношение дает только стоимость оптимального выравнивания образца с текстом, но не последовательность операций редактирования, приводящую к этому выравниванию. Получить список операций можно,

двигаясь в обратном направлении от ячейки $D[m, n]$ в матрице полной стоимости $D$. Чтобы добраться до ячейки $D[m, n]$, мы должны были выйти или из ячейки $D[m-1, n]$ (удаление образца и вставка текста), или из $D[m, n-1]$ (удаление текста и вставка образца) или $D[m-1, n-1]$ (замена/совпадение). Опцию, которая была выбрана фактически, можно выяснить по этим значениям стоимости и по символам $p_m$ и $t_n$. Продолжая двигаться в обратном направлении к предыдущей ячейке, можно восстановить все операции редактирования. Реализация этого алгоритма на языке C приводится в *разд. 10.2*.

- *Как поступить, если обе строки очень похожи друг на друга?*

  Рассмотренный ранее алгоритм динамического программирования заполняет матрицу размером $m \times n$, чтобы вычислить расстояние редактирования. Но для поиска выравнивания, включающего в себя комбинацию из не более чем $d$ вставок, удалений и замен, нужно только обойти полосу из $O(dn)$ ячеек на расстоянии $d$ в каждую сторону от главной диагонали. Если в этой полосе нет выравнивания низкой стоимости, его нет и во всей матрице стоимости.

- *Насколько длинной является строка-образец?*

  Недавно появился новый подход к сравнению строк, в котором используется то обстоятельство, что современные компьютеры могут выполнять операции на словах длиной в 64 бита. В таком машинном слове можно разместить восемь 8-битовых символов ASCII, что стимулирует разработку *алгоритмов с параллелизмом на уровне битов*, в которых за одну операцию выполняется несколько сравнений.

  Основная идея далеко не тривиальна. Для каждой буквы $\alpha$ алфавита создаем битовую маску $B_\alpha$, в которой $i$-й бит $B_\alpha[i]$ равен 1 тогда и только тогда, когда $i$-м символом образца является $\alpha$. Допустим теперь, что имеется такой битовый вектор совпадения $M_j$ для $j$-й позиции в текстовой строке, что $M_j[i] = 1$ тогда и только тогда, когда первые $i$ битов образца точно совпадают с символами текста, начиная с $(j - i + 1)$-го и заканчивая $j$-м. Мы можем найти все биты $M_{i+1}$ посредством лишь двух операций, а именно сдвинув вектор $M_j$ на один бит вправо, а потом выполнив для него побитовую операцию И с маской $B_\alpha$, где $\alpha$ — символ в $(j + 1)$-й позиции текста.

  Такой алгоритм с параллелизмом на уровне битов, обобщенный до нечеткого сравнения, используется в рассматриваемой далее программе agrep. Подобные алгоритмы легко поддаются реализации, а работают они во много раз быстрее алгоритмов динамического программирования.

- *Как минимизировать требования к памяти?*

  Квадратичный объем памяти, требуемый для хранения таблицы динамического программирования, представляет проблему, более серьезную, чем время исполнения соответствующих алгоритмов. К счастью, вычисление $D[m, n]$ требует только $O(\min(m, n))$ памяти. И для вычисления оптимальной стоимости нужно сопровождать лишь две активные строки (или столбца) матрицы. Вся матрица требуется только для воссоздания последовательности операций, в результате которых было получено то или иное выравнивание.

Но для эффективного вычисления за линейное время оптимального выравнивания можно использовать рекурсивный алгоритм Хиршберга. За первый проход описанного ранее алгоритма с линейной сложностью по памяти для вычисления $D[m, n]$ можно выяснить, какая ячейка среднего элемента $D[m/2, x]$ была использована для оптимизации $D[m, n]$. Это сводит нашу задачу к задаче поиска наилучших путей от $D[1, 1]$ до $D[x/2, x]$ и от $D[m/2, x]$ до $D[m/2, n]$, причем оба пути могут быть найдены рекурсивно. В каждом следующем проходе мы исключаем из рассмотрения половину элементов матрицы из предыдущего прохода, поэтому общее время остается равным $O(mn)$. Этот алгоритм с линейной сложностью по памяти обладает хорошей производительностью на длинных строках.

- *Следует ли по-разному оценивать многократные повторы операций вставки и удаления?*

  Во многих приложениях сравнения строк оказывается предпочтение выравниваниям, в которых операции вставки и/или удаления сгруппированы в небольшое количество последовательностей. Удаление абзаца из текста, по идее, должно стоить меньше, чем удаление соответствующего количества разбросанных одиночных символов, поскольку при этом выполняется одно (хотя и значительное) изменение.

  Применение штрафов за появление разрывов при сравнении строк позволяет корректно учитывать такие изменения. Обычно для каждой операции вставки/удаления $t$ последовательных символов устанавливается стоимость $A + Bt$, где $A$ — стоимость создания разрыва, а $B$ — стоимость удаления одного символа. Если значение $A$ велико по сравнению со значением $B$, то при выравнивании появится стимул выполнять относительно небольшое количество последовательных операций удаления.

  Сравнение строк при таких *аффинных* штрафах за разрывы можно выполнять за квадратичное время, как и обычное вычисление расстояния редактирования. Мы будем использовать отдельные рекуррентные соотношения $E$ и $F$ для вычисления стоимости операций вставки и удаления в режиме разрыва, чтобы платить только один раз за создание разрыва:

$$V(i,j) = \max(E(i,j), F(i,j), G(i,j))$$
$$G(i,j) = V(i-1, j-1) + match(i,j)$$
$$E(i,j) = \max(E(i,j-1), V(i,j-1) - A) - B$$
$$F(i,j) = \max(F(i-1,j), V(i-1,j) - A) - B$$

  При постоянном времени обращения к каждой ячейке время исполнения этого алгоритма равно $O(mn)$.

- *Считаются ли похожими строки с одинаковым произношением?*

  Для некоторых приложений лучше подходят другие модели нечеткого сравнения строк. Особый интерес представляет схема хеширования Soundex, в которой одинаково звучащим словам присваивается одинаковый индекс. Эта схема может быть полезной при проверке, не являются ли два по-разному написанных слова в действительности одним и тем же словом. Например, мою фамилию пишут как «Skina», «Skinnia», «Schiena», а иногда и «Skiena». Всем этим по-разному написанным словам присваивается одинаковый индекс Soundex — S25.

Алгоритм Soundex отбрасывает гласные, непроизносимые и повторяющиеся буквы, а потом назначает оставшимся буквам числа в соответствии со следующими классами: BFPV — 1, CGJKQSXZ — 2, DT — 3, L — 4, MN — 5, R — 6. Буквам HWY не присваиваются никакие цифры. Код слова состоит из первой буквы имени, за которой следуют максимум три цифры. Хотя такой подход кажется не очень естественным, на практике он работает достаточно хорошо. А практика уже весьма продолжительная — Soundex используется с 1920-х годов.

### Реализации

Для нечеткого сравнения строк существует несколько отличных программных средств. Программа agrep (см. [WM92a, WM92b]) поддерживает поиск в тексте с орфографическими ошибками. Текущую версию этой программы можно загрузить с веб-страницы **http://www.tgries.de/agrep**. Программа nrgrep (см. [Nav01b]) сочетает параллелизм на уровне битов с фильтрацией и имеет постоянное время исполнения, хотя не всегда работает быстрее, чем agrep. Загрузить программу можно с веб-сайта **https://www.dcc.uchile.cl/~gnavarro/software/**.

Библиотека TRE поиска совпадений с регулярными выражениями предназначена для нахождения точных и приблизительных совпадений и обладает более широкими возможностями, чем программа agrep. Временная сложность программы в худшем случае равна $O(nm^2)$, где $m$ — длина списка используемых регулярных выражений. Загрузить программу можно с веб-сайта **at https://github.com/laurikari/tre/**.

В Wikipedia можно найти программы для вычисления расстояния редактирования — в частности, расстояния Левенштейна, написанные на разных языках (включая Ada, C++, Emacs Lisp, JavaScript, Java, PHP, Python, Ruby VB и C#). Дополнительную информацию можно найти на веб-сайте **http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance**

#### *Примечания*

В последнее время наблюдаются определенные успехи в области нечеткого сравнения строк, особенно в алгоритмах, применяющих параллелизм на уровне битов. Алгоритмам обработки строк посвящены книги [CHL07] и [Gus97], а самым лучшим справочником по методам сравнения строк является [NR07].

Считается, что базовый алгоритм динамического программирования для получения выравниваний впервые описан в работе [WF74], хотя, по-видимому, он был известен и раньше. Широта диапазона применения нечеткого сравнения строк была продемонстрирована в книге [SK99], которая по сей день является полезным историческим справочником в этой области. Обзоры решений задачи нечеткого сравнения строк представлены в работах [HD80] и [Nav01a]. Алгоритм Хиршберга с линейной сложностью по памяти (см. [Hir75]) рассматривается в книгах [CR03] и [Gus97].

В работе [MP80] приведен алгоритм для вычисления расстояния редактирования между строками длиной в $m$ и $n$ символов за время $O(mn/\log(\min\{m, n\}))$ для алфавитов постоянного размера. В алгоритме использованы идеи из *алгоритма четырех русских* для умножения булевых матриц (см. [ADKF70]). В работе [BI15] приводится результат недавних исследований сложности, показывающий, что расстояние редактирования не поддается вычислению за время $O(n^{2-\varepsilon})$ без нарушения сильной гипотезы об экспоненциальном времени. Другой недавний прорыв — алгоритмы с субквадратичным временем выполнения,

которые аппроксимируют расстояние редактирования с точностью до постоянного коэффициента (см. [CDG+18]).

Формулировка задачи нечеткого сравнения строк в терминах поиска кратчайшего пути позволяет создать множество алгоритмов, дающих хорошие результаты для небольших расстояний редактирования, включая алгоритмы с временем исполнения $O(n \lg n + d^2)$ (см. [Mye86]) и $O(dn)$ (см. [LV88])

Максимальную возрастающую последовательность можно вычислить за время $O(n \lg n)$ (см. [HS77]), как описано в [Man89].

В числе алгоритмов нечеткого сравнения строк, применяющих параллелизм на уровне битов, можно назвать алгоритм Майерса (Myers) (см. [Mye99b]) с временем исполнения $O(mn/w)$, где $w$ — количество битов компьютерного слова. Результаты экспериментальных исследований алгоритмов с параллелизмом на уровне битов представлены в работах [FN04], [HFN05] и [NR,00].

Система Soundex была изобретена и запатентована Оделл (M. K. Odell) и Расселлом (R. C. Russell). Описание этой системы можно найти в работах [BR95] и [Knu98]. Современная разработка Metaphone (см. [BR95] и [Par90]) представляет собой попытку создания системы, превосходящей Soundex. Применение таких систем фонетического хеширования для унификации названий задач рассматривается в работе [LMS06].

### Родственные задачи

Поиск строк (см. *разд. 21.3*), поиск максимальной общей подстроки (см. *разд. 21.8*).

## 21.5. Сжатие текста

**Вход.** Текстовая строка $S$.

**Задача.** Преобразовать строку $S$ в более короткую строку $S'$, из которой можно правильно воссоздать исходную строку $S$ (рис. 21.6).

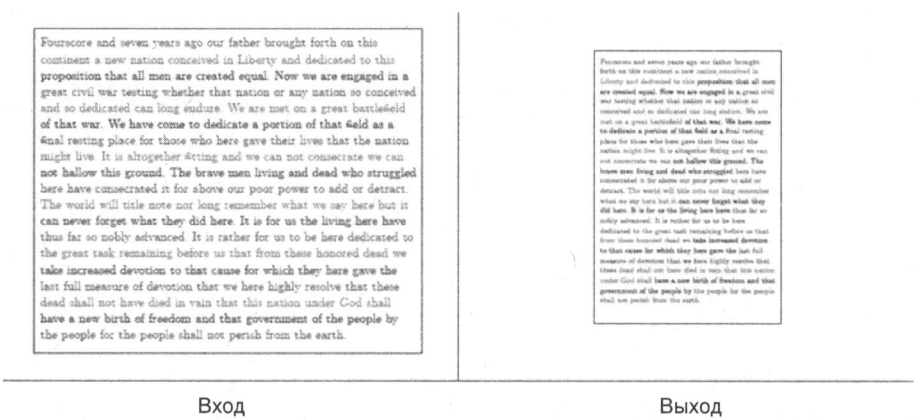



**Рис. 21.6.** Сжатие текста

### Обсуждение

Хотя емкость внешних устройств хранения данных удваивается каждый год, ее все время не хватает. Снижение цен на устройства хранения ничуть не уменьшило интерес

к сжатию данных, вероятно, потому, что данных, подлежащих сжатию, становится все больше. *Сжатие данных* — это алгоритмическая задача, в которой требуется найти экономичную кодировку для файла указанного типа. Развитие компьютерных сетей поставило новую цель перед задачей сжатия данных — повышение эффективной пропускной способности за счет уменьшения количества передаваемых битов.

Создается впечатление, что разработчикам *нравится* изобретать специальные методы сжатия данных для каждого конкретного приложения. Иногда производительность этих специализированных методов даже превосходит производительность методов общего назначения, но обычно это не так. При выборе правильного алгоритма сжатия возникает несколько вопросов.

- *Требуется ли точное восстановление исходных данных после сжатия?*

  Основной вопрос, возникающий при сжатии данных, заключается в том, выполнять сжатие *с потерями* или *без потерь*. Приложения для хранения документов обычно требуют обратимого сжатия (без потерь), поскольку пользователи не хотят, чтобы их данные подвергались изменениям. При сжатии изображений или видеофайлов точность данных не настолько важна, т. к. небольшие погрешности незаметны для зрителя. Применение сжатия с потерями позволяет получить значительно более высокую степень сжатия, вследствие чего этот тип сжатия применяется в большинстве приложений для сжатия аудио- и видеоданных и изображений.

- *Можно ли упростить данные перед сжатием?*

  Самый эффективный способ увеличения свободного дискового пространства — удаление ненужных файлов. Аналогичным образом любая предварительная обработка, уменьшающая объем информации в файле, приводит к более эффективному сжатию. Постарайтесь выяснить, можно ли удалить из файла избыточные пробельные символы, преобразовать все буквы текста в заглавные или снять форматирование.

  Особый интерес представляет упрощение данных с помощью *преобразования Барроуза — Уилера*. По ходу этого преобразования выполняется сортировка всех $n$ циклических сдвигов $n$-символьной входной строки, а затем возвращается последний символ каждого сдвига. Например, строка *abab* имеет следующие циклические сдвиги: *abab*, *baba*, *abab* и *baba*. После сортировки получаем строки *abab*, *abab*, *baba* и *baba*. Считывая последний символ каждой из этих строк, получаем результат преобразования — *bbaa*.

  При условии, что последний символ входной строки уникален (например, представляет собой символ конца строки), это преобразование является полностью обратимым. Строка, предварительно обработанная с помощью преобразования Барроуза — Уилера, сжимается на 10–15% лучше, чем первоначальный текст, поскольку повторяющиеся слова превращаются в блоки повторяющихся символов. Кроме того, это преобразование можно осуществить за линейное время.

- *Что делать, если алгоритм сжатия запатентован?*

  Некоторые алгоритмы сжатия данных были запатентованы. Одним из самых известных примеров является рассматриваемая далее версия LZW алгоритма Лемпеля — Зива. К счастью, в настоящее время срок действия этого патента истек, так же

как и алгоритма сжатия JPEG. Обычно у любого алгоритма сжатия имеются версии, которыми можно пользоваться без ограничений, причем работают они так же хорошо, как и запатентованный вариант. Но когда запатентованный алгоритм затрагивает область какого-либо популярного стандарта, то это вызывает проблемы.

- *Как сжимать изображения?*

  Самым простым обратимым алгоритмом сжатия изображений является *кодирование длин серий*. При таком способе сжатия последовательности одинаковых пикселов (серии) заменяются одним экземпляром пиксела и числом, указывающим длину последовательности. Этот метод хорошо работает с двоичными представлениями изображений, имеющими большие области одинаковых пикселов, — например, с отсканированным текстом. Но на изображениях, имеющих много градаций цвета и/или содержащих шум, производительность падает. Сильное влияние на качество сжатия оказывают два фактора: выбор размера поля, содержащего длину последовательности, и выбор порядка обхода при преобразовании двумерного изображения в поток пикселов.

  Для профессиональных приложений сжатия аудиоданных, видеоданных и изображений я рекомендую использовать один из популярных методов сжатия с потерями и не пытаться реализовать свой собственный. Стандартным высокопроизводительным методом сжатия изображений признан JPEG, а MPEG предназначен для сжатия видеоданных.

- *Должно ли сжатие выполняться в реальном времени?*

  Быстрое восстановление данных нередко оказывается важнее, чем их быстрое сжатие. Например, видеоданные на YouTube сжимаются только один раз, но восстанавливаются при каждом его воспроизведении. Впрочем, существует и противоположный пример: операционной системе для увеличения эффективной емкости диска посредством сжатия файлов требуется симметричный алгоритм, обладающий также быстрым временем сжатия.

Существуют десятки алгоритмов для сжатия текста, но их можно разбить на две отдельные группы. *Статические алгоритмы*, такие как метод Хаффмана, создают одну кодовую таблицу, исследуя полностью весь документ. *Адаптивные алгоритмы*, такие как алгоритм Лемпеля — Зива, динамически создают кодовую таблицу, которая адаптируется под локальное распределение символов документа. В большинстве случаев следует предпочитать адаптивный алгоритм, но и статические алгоритмы могут представлять интерес.

- *Код Хаффмана.*

  Алгоритм Хаффмана заменяет каждый символ алфавита кодовой строкой битов переменной длины. Использование восьми битов для кодирования каждой буквы неэкономично, поскольку некоторые символы (например, буква «е») встречаются намного чаще, чем другие (например, буква «q»). При кодировании сжатия методом Хаффмана букве «е» присваивается короткий код, а букве «q» — более длинный.

  Код Хаффмана можно создать, используя «жадный» алгоритм. Сначала сортируем символы в порядке возрастания частоты их вхождения в текст. Два самых редких символа $x$ и $y$ объединяются в один новый символ $xy$ с частотой вхождения, равной

сумме двух его исходных символов. В результате мы получаем меньший набор символов. Эта операция повторяется $n - 1$ раз, пока все символы не будут слиты попарно. Посредством таких операций объединения строится корневое двоичное дерево, в котором исходные символы алфавита представлены листьями. Биты слова двоичного кода для каждого символа определяются выбором правого или левого пути в проходе от корня к листу. При написании программы упорядочивания символов по частоте использования можно применять очереди с приоритетами, что позволит выполнять эту операцию за время $O(n \lg n)$.

Несмотря на свою популярность, алгоритм Хаффмана имеет три недостатка. Во-первых, для того чтобы закодировать документ, по нему необходимо выполнить два прохода: первый, чтобы создать таблицу кодирования, а второй, чтобы выполнить само кодирование текста. Во-вторых, таблицу кодирования нужно явно сохранять вместе с закодированным документом, чтобы можно было декодировать его, и это приводит к неэкономному расходу памяти при кодировании коротких документов. Наконец, алгоритм Хаффмана эффективен при неравномерном распределении символов, в то время как адаптивные алгоритмы способны распознавать повторяющиеся последовательности — такие как, например, *0101010101*.

- *Алгоритмы Лемпеля — Зива.*

  Эти алгоритмы (включая популярный вариант LZW) сжимают текст, создавая таблицу кодирования по ходу чтения документа. Получаемая таблица корректируется по мере перемещения по тексту. Благодаря использованию хорошо продуманного протокола программы кодирования и декодирования работают с одной и той же таблицей, что позволяет избежать потери информации.

  Алгоритмы Лемпеля — Зива создают кодовые таблицы часто встречающихся подстрок, размер которых может быть очень большим. В результате алгоритмы этого типа могут использовать часто встречающиеся слоги, слова и фразы, чтобы выполнить кодирование наилучшим образом. Алгоритмы приспосабливаются к локальным изменениям в распределении элементов текста, что является важным, поскольку многие документы обладают значительной пространственной локальностью.

  Замечательным свойством алгоритмов Лемпеля — Зива является их устойчивость при обработке данных разных типов. Разработать для конкретного приложения собственный специализированный алгоритм, превосходящий алгоритм Лемпеля — Зива, довольно трудно. Так что я рекомендую не предпринимать таких попыток. Если имеется возможность устранить специфическую для приложения избыточность посредством простой предварительной обработки, воспользуйтесь ею. Но не тратьте понапрасну время, пытаясь создать собственный алгоритм сжатия. Вряд ли вам удастся получить значительно лучшие результаты, чем выдает программа gzip или другая распространенная утилита.

### Реализации

Самой популярной программой сжатия текста, вероятно, является программа gzip, которая реализует публично доступный вариант алгоритма Лемпеля — Зива. Эта программа распространяется с лицензией GNU, и загрузить ее можно с веб-сайта **https://www.gzip.org**.

Всегда приходится искать разумный компромисс между коэффициентом сжатия и временем работы алгоритма. Альтернативу программе gzip составляет программа сжатия bzip2, в которой используется преобразование Барроуза — Уилера. Она обеспечивает лучший коэффициент сжатия, чем программа gzip, но за счет увеличения времени исполнения. Создатели некоторых алгоритмов впадают в крайность, жертвуя скоростью ради повышенной компактности. Типичные представители таких алгоритмов собраны на веб-сайте **http://mattmahoney.net/dc/**. Лично я верю, что для тех людей, которые присылают мне файлы, упакованные странными программами сжатия, зарезервировано особое место в аду. Я всегда удаляю такие файлы и требую, чтобы они прислали материал заново, упакованный посредством программы gzip.

Авторитетное сравнение программ сжатия, включающее в себя ссылки на все доступное программное обеспечение, представлено на веб-сайте **http://www.maximumcompression.com/**.

#### *Примечания*

Существует большое количество книг, посвященных сжатию данных. Из недавно вышедших можно назвать книги [Say17] и [Sal06]. Также могу порекомендовать и уже не новую работу [BCW90]. Обзор алгоритмов сжатия текстов приведен в работах [CL98] и [KA10].

Хорошее описание алгоритма Хаффмана (см. [Huf52]) можно найти в книгах [AHU83] и [CLRS09]. Алгоритм Лемпеля — Зива и его варианты описаны в работах [Wel84] и [ZL78]. Преобразование Барроуза — Уилера представлено в работе [BW94].

Главным форумом по обмену информацией в области сжатия данных является ежегодная тематическая конференция Института IEEE (подробности см. на веб-сайте **https://www.cs.brandeis.edu/~dcc**). Сжатие данных — хорошо развитая техническая область, и в последнее время усилия разработчиков сосредоточены на весьма незначительных улучшениях алгоритмов, особенно в случае сжатия текстовых данных. Однако радует то обстоятельство, что ежегодная конференция IEEE проводится на горнолыжном курорте мирового класса в штате Юта.

### Родственные задачи

Поиск минимальной общей надстроки (см. *разд. 21.9*), криптография (см. *разд. 21.6*).

## 21.6. Криптография

**Вход.** Открытое сообщение $T$ или зашифрованный текст $E$ и ключ $k$.

**Задача.** С помощью ключа $k$ зашифровать сообщение $T$ (расшифровать текст $E$), получив в результате зашифрованный текст $E$ (открытое сообщение $T$) (рис. 21.7).

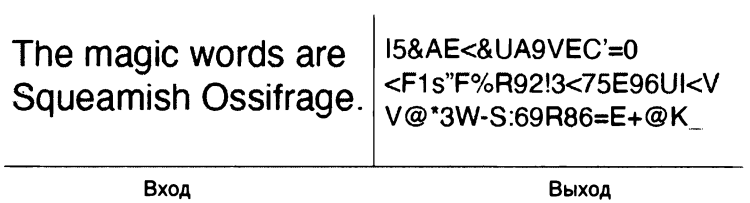



**Рис. 21.7.** Пример шифрования текста

### Обсуждение

С учетом того, что широкое распространение компьютерных сетей предоставляет все больше возможностей несанкционированного доступа к конфиденциальной информации, важную роль в сохранении ее целостности играет криптография. Она повышает безопасность обмена сообщениями, позволяя обрабатывать их таким образом, что их будет невозможно прочесть, если они попадут в чужие руки. Хотя этой дисциплине по крайней мере две тысячи лет, ее алгоритмические и математические основы лишь недавно сформировались настолько, что появилась возможность рассуждать о доказуемо безопасных системах шифрования.

Идеи и приложения шифрования выходят далеко за рамки задач собственно «шифрования» и «дешифрования». Сейчас эта область включает такие математические конструкции, как криптографические хеши, цифровые подписи, а также базовые протоколы, предоставляющие гарантии безопасности.

Существуют три класса систем шифрования:

- *шифр Цезаря.*

  Самые древние шифры построены на замене каждого символа алфавита другим символом того же алфавита. В наиболее слабых шифрах выполняется просто циклический сдвиг алфавита на заданное количество символов (скажем, 13), вследствие чего они имеют только 26 возможных ключей[3]. Лучше было бы использовать произвольную перестановку букв, что дает уже 26! возможных ключей. Но даже и в этом случае такие системы шифрования можно с легкостью взломать, подсчитав частоту появления каждого символа в тексте и применив знание частотного распределения букв в текстах используемого алфавита. И хотя существуют варианты этого шифра, более устойчивые к взлому, ни один из них не будет таким надежным, как рассматриваемые далее шифры AES или RSA;

- *блочные шифры.*

  В шифрах этого типа биты текста многократно перемешиваются в соответствии с заданным ключом. Классическим примером такого шифра является шифр Data Encryption Standard (DES). Он был одобрен в 1976 году в качестве Федерального стандарта США для обработки информации, но в 2005 году официально утратил этот статус, и его заменили более сильным шифром Advanced Encryption Standard (AES). Простой вариант шифра DES под названием «тройной DES» позволяет осуществлять шифрование с помощью ключа длиной в 112 битов за счет троекратного применения шифра DES с двумя 56-битовыми ключами. Тем не менее со временем и этот шифр также стал уязвимым и в 2018 году был объявлен устаревшим с запретом на его дальнейшее использование после 2023 года. А 256-битовый шифр AES все еще считается очень надежным. Библиотеки с реализацией этого шифра существуют для всех основных языков программирования, включая C/C++, Java, JavaScript и Python. На этом шифре основана безопасность WhatsApp, Facebook Messenger и многих других систем;

---

[3] Речь здесь идет об английском алфавите. — *Прим. пер.*

- *шифрование с открытым ключом.*

  Если вы боитесь, что какие-либо недоброжелатели читают ваши сообщения, вы должны держать в секрете ключ, необходимый для их расшифровки. Однако в системах с открытым ключом для шифрования и дешифрования сообщений используются разные ключи. Так как ключ шифрования бесполезен для расшифровки, он может быть открытым без риска для безопасности. Решение о распределении ключей лежит в основе успеха шифрования с открытым ключом.

  Классический пример такой системы шифрования представляет собой система RSA, названная так в честь ее изобретателей: Ривеста (Rivest), Шамира (Shamir) и Адлемана (Adleman). Безопасность системы RSA основана на вычислительной сложности задач разложения на множители и проверки чисел на простоту (см. *разд. 16.8*). Процесс шифрования протекает сравнительно быстро, поскольку для создания ключа используется проверка на простоту, а сложность расшифровки вытекает из сложности разложения на множители. Сейчас это основание находится под все нарастающей угрозой прогресса в области квантовых вычислений. Впрочем, на момент работы над материалом этого издания самым большим целым числом, неоспоримо разложенным на множители на квантовом компьютере посредством алгоритма Шора, является число 15, так что пока это не близкая угроза. Тем не менее стоит не упускать ее из вида.

  К сожалению, система шифрования RSA по сравнению с другими системами работает медленно. В частности, она примерно в 100–1000 раз медленнее, чем шифр DES. Тем не менее она отрабатывает свой хлеб, делая возможным шифрование с открытым ключом.

Важнейший фактор, влияющий на выбор системы шифрования, — это требуемый уровень безопасности. От кого вы пытаетесь защитить свою переписку: от вашей бабушки, квартирных воров, организованной преступности или спецслужб? Используя современную реализацию шифров AES или RSA, вы сможете чувствовать себя в безопасности, по крайней мере, в ближайшее время. Однако постоянно растущая мощность компьютеров способна на удивление быстро расшатать криптосистему, как упоминалось ранее. Поэтому обязательно используйте ключи максимальной длины и следите за новостями в области шифрования, если вы планируете долгосрочное хранение конфиденциальной информации.

Что касается меня лично, признаюсь, что я использую DES для шифрования задач выпускного экзамена в конце каждого семестра. Этого шифра оказалось более чем достаточно, когда один решительно настроенный студент проник в мой офис в поисках задач. Результат был бы иным, если бы взломщиком оказался агент спецслужб. При этом важно понимать, что *самые серьезные бреши в безопасности обусловлены человеческим фактором, а не качеством алгоритма.* Придумать достаточно длинный и трудный для отгадывания пароль и не записать его на бумажке, приклеенной к монитору, гораздо важнее, чем излишне усердствовать в выборе алгоритма шифрования.

При одинаковой длине ключа большинство симметричных систем шифрования труднее поддаются взлому, чем системы с открытым ключом. Это означает, что для симметричного шифрования можно использовать намного более короткие ключи, чем для шифрования с открытым ключом. Институт NIST и лаборатория RSA предоставляют

списки рекомендуемых размеров ключей для безопасного шифрования, и на момент подготовки этого издания они рекомендуют 256-битовые симметричные ключи как эквивалентные 15 360-битовым асимметричным. Эта разница объясняет, почему алгоритмы шифрования симметричным ключом работают на несколько порядков быстрее алгоритмов шифрования с ключом открытым.

Простые шифры вроде шифра Цезаря легко программируются. По этой причине их можно использовать в приложениях, требующих минимального уровня безопасности, — например, для того, чтобы скрыть ответы на загадки. Но поскольку эти шифры легко поддаются взлому, на них никогда не следует полагаться в приложениях, требующих серьезного уровня безопасности.

А еще *никогда* не следует пытаться разработать собственную систему шифрования. Безопасность шифров AES и RSA общепризнана, т. к. они выдержали многолетнее испытание массовым использованием. За это время было предложено много других систем шифрования, которые оказались уязвимыми к взлому. Разработкой алгоритмов шифрования должны заниматься только профессионалы. Если же вам поручили создать систему безопасности, начните с изучения какой-либо признанной программы — например, PGP, чтобы разобраться, как в ней решаются вопросы выбора и распространения ключей. Стойкость любой системы безопасности определяется ее самым слабым звеном.

На практике часто возникают следующие вопросы, связанные с шифрованием.

- *Как проверить, что данные не были повреждены случайно?*

  Часто требуется удостовериться в том, что при передаче данных они не подверглись случайным искажениям и полученные данные идентичны отправленным. Одно из решений такой задачи — передать полученные данные обратно отправителю для подтверждения идентичности обоих текстов. Но этот метод не срабатывает, если при обратной передаче происходят противоположные ошибки. К тому же при такой схеме канал передачи данных задействуется в обоих направлениях, вследствие чего его пропускная способность уменьшается вдвое, что представляет серьезное неудобство.

  Более эффективный метод — использование контрольной суммы, т. е. хеширование длинного текста в небольшое число и передача этой контрольной суммы вместе с текстом. Принимающая сторона вычисляет контрольную сумму принятого текста и сравнивает ее с первоначальной. В самой простой схеме с контрольной суммой вычисляется сумма байтов текста по модулю некоторой константы — например, $2^8 = 256$. Но такая схема не выявляет ошибку, при которой несколько символов меняются местами, поскольку сложение коммутативно.

  Более надежный способ вычисления контрольной суммы заключается в использовании *циклического избыточного кода*. Он применяется в большинстве систем связи и в компьютерах для проверки целостности обмена данных с диском. Для получения этой контрольной суммы вычисляется остаток от деления двух многочленов, где делитель является функцией входного текста. Разработка этих многочленов является сложной математической задачей, но такая контрольная сумма обеспечивает обнаружение всех достаточно вероятных ошибок.

- *Как проверить, что данные не были повреждены преднамеренно?*

  Метод контроля с помощью циклического избыточного кода хорошо распознает случайные искажения, но не годится для выявления преднамеренных. Для этих целей лучше подходят криптографические функции хеширования — такие как MD5 или SHA-256, которые легко вычисляются для того или иного документа, но являются труднообратимыми. Это означает, что по имеющемуся хеш-значению $x$ трудно воссоздать такой документ $d$, для которого $H(d) = x$. Указанное свойство хеш-функций делает их полезными для использования в качестве цифровых подписей и в других приложениях.

- *Как доказать, что файл не был изменен?*

  Если я вышлю вам контракт в электронной форме, ничто не помешает вам изменить полученный файл и утверждать, что ваша версия отражает нашу договоренность. Мне нужен способ доказать, что модифицированная версия документа является подделкой. Для этого служит *цифровая подпись* — криптографический способ подтверждения подлинности документа, играющий важную роль в поддержании целостности блокчейнов, используемых в криптовалютах, — в том же, например, Bitcoin.

  Я могу вычислить контрольную сумму для любого файла и зашифровать ее с помощью закрытого ключа. Вместе с файлом контракта я высылаю и его зашифрованную контрольную сумму. Конечно же, вы можете отредактировать файл, но, чтобы обмануть суд, вам надо будет отредактировать и зашифрованную контрольную сумму так, чтобы после ее расшифровки получилась исходная. А вот создание другого файла с такой же контрольной суммой — задача невыполнимая. Для полной безопасности мне понадобятся услуги пользующейся всеобщим доверием третьей стороны, которая удостоверит подлинность цифровой подписи и свяжет закрытый ключ со мной.

- *Как ограничить доступ к материалам, защищенным авторским правом?*

  Важное новое применение шифрования — это управление авторскими правами на цифровые аудио- и видеоматериалы. Принципиальным вопросом здесь является скорость декодирования, чтобы она соответствовала скорости пересылки данных в реальном времени. Такие *потоковые шифры* обычно включают в себя генерирование потока псевдослучайных битов с использованием, например, генератора кода на регистре сдвига. Операция исключающего ИЛИ над этими битами и потоком данных дает зашифрованную последовательность. Для восстановления исходных данных опять выполняется операция исключающего ИЛИ над зашифрованным потоком данных с тем же самым потоком псевдослучайных битов, который использовался для шифрования.

  Впрочем, было доказано, что высокоскоростные системы потокового шифрования сравнительно легко поддаются взлому. Так что современное решение этой проблемы заключается в принятии специальных законов и уголовном преследовании за их нарушение.

### Реализации

Криптографическая библиотека Nettle содержит среди прочего функции хеширования MD5 и SHA-256, а также блочные шифры DES, AES и некоторые другие. В библиотеку включена и реализация системы RSA. Загрузить библиотеку Nettle можно по адресу **https://www.lysator.liu.se/~nisse/nettle/**. А на странице **http://csrc.nist.gov/groups/ST/toolkit** Институт NIST представляет коллекцию стандартов и руководств по разработке алгоритмов шифрования.

Большая библиотека схем шифрования Crypto++ классов языка C++ включает все рассмотренные в этом разделе системы шифрования. Загрузить библиотеку можно с веб-сайта **https://www.cryptopp.com**.

Многие популярные утилиты с открытым кодом используют профессиональные схемы шифрования и служат хорошими моделями текущей практики в этой области. Программу GnuPG — версию с открытым кодом программы PGP — можно загрузить по адресу **https://www.gnupg.org/**. А программу OpenSSL для аутентификации доступа к компьютерным системам — по адресу **https://www.openssl.org/**.

Библиотека Boost CRC Library содержит несколько реализаций алгоритмов вычисления контрольной суммы с помощью циклического избыточного кода. Загрузить библиотеку можно с веб-сайта **https://www.boost.org/libs/crc/**.

#### *Примечания*

В книге [MOV96] приведено описание технических деталей всех аспектов криптографии. Электронную версию книги можно найти на веб-сайте **http://www.cacr.math.uwaterloo.ca/hac/**. В книге [Sch15] содержится всестороннее обсуждение разных алгоритмов шифрования, но, пожалуй, книга [FS03] предлагает лучшее введение в эту область. В книге [Kah67] увлекательно изложена история криптографии, начиная с древних времен до 1967 года.

Обсуждение алгоритма RSA (см. [RSA78]) приводится, в частности, в книге [CLRS09]. Обширная информация о нем представлена на домашней странице RSA по адресу **http://www.rsa.com/rsalabs/**. Квантовые технологии дали толчок к разработке новых методов шифрования (см. [BB14]), а также новых способов взламывания уже существующих шифров. Криптография в эпоху квантовых вычислений является областью активных исследований (см. [CJL+16]).

Главным источником информации о современном состоянии криптографии является Агентство национальной безопасности США (National Security Agency, NSA). История шифра DES хорошо описана в книге [Sch15]. Особенно спорным представляется решение NSA ограничить длину ключа 56 битами.

Хеш-функция MD5 (см. [Riv92]) используется в программе PGP для вычисления цифровых подписей. Ее описание можно найти, например, в [Sch15] и [Sta06]. Кстати, с безопасностью этой функции были выявлены некоторые проблемы (см. [WY05]). Семейство хеш-функций SHA (в частности, функции SHA-256 и SHA-512) производит более надежное впечатление.

### Родственные задачи

Разложение на множители и проверка чисел на простоту (см. *разд. 16.8*), сжатие текста (см. *разд. 21.5*).

# 21.7. Минимизация конечного автомата

**Вход.** Детерминированный конечный автомат $M$.

**Задача.** Создать наименьший детерминированный конечный автомат $M'$, поведение которого идентично поведению автомата $M$ (рис. 21.8).

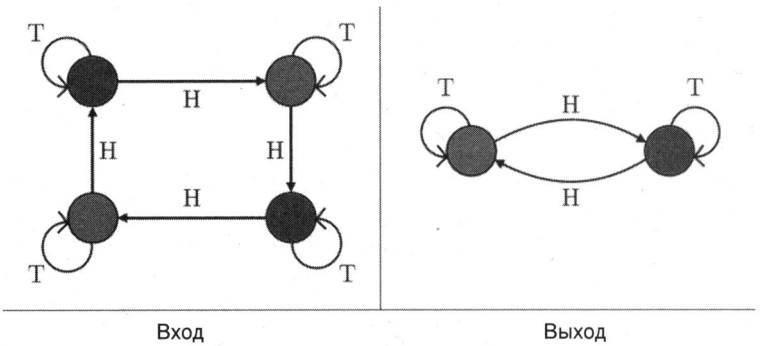



**Рис. 21.8.** Пример минимизации конечного автомата

### Обсуждение

Конечные автоматы широко используются в приложениях, связанных с распознаванием образов. Современные языки программирования, такие как Java и Python, содержат встроенные средства поддержки *регулярных выражений*, являющихся самым естественным способом определения автоматов. Конечные автоматы часто применяются в системах управления и компиляторах для кодирования текущего состояния и возможных действий или переходов. Минимизация размера автоматов понижает как затраты по хранению данных, так и стоимость исполнения при использовании таких автоматов.

Конечные автоматы определяются посредством ориентированных графов. Каждая вершина представляет состояние, а каждое помеченное символом алфавита ребро определяет переход из одного состояния в другое при получении этого символа. Показанные на рис. 21.8 автоматы анализируют последовательность бросков монеты. Закрашенные вершины обозначают состояние, при котором «орел» выпал четное количество раз. Такие автоматы можно представить, используя граф (см. *разд. 15.4*) или *матрицу переходов* размером $n \times |\Sigma|$, где $\Sigma$ — размер *алфавита* символов автомата. Выражение $M[i, j]$ представляет состояние, в которое был выполнен переход из состояния $i$ при получении символа $j$.

Конечные автоматы часто используются для поиска образцов, представленных в виде регулярных выражений, которые являются результатом выполнения операций И, ИЛИ или повтора над регулярными выражениями меньшего размера. Например, регулярное выражение $a(a + b + c)^*a$ соответствует любой строке алфавита ($a$, $b$, $c$), которая начинается и заканчивается на букву $a$. Самый лучший способ проверки, описывается ли входная строка регулярным выражением $R$, заключается в создании конечного автомата, эквивалентного $R$, с последующей эмуляцией этого автомата на проверяемой

строке. Альтернативные подходы к решению задачи поиска строк рассматриваются в *разд. 21.3*.

Здесь мы рассмотрим три задачи, связанные с конечными автоматами:

- *минимизация детерминированных конечных автоматов.*

  У сложных конечных автоматов матрицы переходов могут становиться недопустимо большого размера, что вызывает необходимость в более «плотных» структурах. Самый простой подход к решению этой проблемы — устранение повторяющихся состояний автомата. Как показано на рис. 21.8, автоматы самого разного размера могут выполнять одну и ту же функцию.

  Алгоритмы минимизации количества состояний детерминированных конечных автоматов можно найти в любом учебнике по теории автоматов. Основной подход к решению этой задачи — приблизительное разбиение состояний на классы эквивалентности с последующим уточнением полученного разбиения. Сначала состояния разделяются на три класса: допускающие, отвергающие и нетерминальные. При этом переходы из каждого узла ведут к выбранному классу по заданному символу. Каждый раз, когда два состояния: $s$ и $t$ — из одного и того же класса $C$ переходят в состояния из разных классов, класс $C$ нужно разбить на два подкласса, один из которых содержит состояние $s$, а другой — состояние $t$.

  Описанный алгоритм перебирает все классы, проверяя необходимость нового разбиения, и, если такая необходимость выявляется, повторяет процесс с начала. Время исполнения этого алгоритма равно $O(n^2)$ для алфавитов постоянного размера, поскольку нужно будет выполнить самое большее $n - 1$ проходов. Полученные после последнего прохода классы эквивалентности соответствуют состояниям в минимальном конечном автомате. Впрочем, существует и более эффективный алгоритм с временем исполнения $O(n \log n)$;

- *преобразование недетерминированных автоматов в детерминированные.*

  С детерминированными конечными автоматами очень легко работать, поскольку в любой момент времени автомат находится в одном-единственном состоянии. *Недетерминированные конечные автоматы* могут находиться в нескольких состояниях одновременно, поэтому их текущее «состояние» представляет подмножество всех возможных состояний автомата.

  Однако любой недетерминированный конечный автомат можно механически преобразовать в эквивалентный детерминированный конечный автомат, который потом удастся минимизировать, как описано ранее. Это преобразование может вызвать экспоненциальный рост количества состояний, хотя и не исключено, что оно уменьшится во время минимизации детерминированного конечного автомата. Такой экспоненциальный «взрыв» делает большинство задач минимизации недетерминированных конечных автоматов PSPACE-полными, а это даже хуже, чем NP-полнота.

  Доказательства эквивалентности недетерминированных конечных автоматов, детерминированных конечных автоматов и регулярных выражений достаточно элементарны и рассматриваются в университетских курсах теории автоматов. Впрочем, при кодировании иногда возникают неожиданные проблемы;

- *создание автоматов на основе регулярных выражений.*

  Для преобразования регулярного выражения в эквивалентный конечный автомат существуют два подхода, причем недетерминированные конечные автоматы легче создавать, чем детерминированные автоматы, но их сложнее эмулировать.

  - Для создания *недетерминированных* конечных автоматов используются ε-переходы, которые являются дополнительными переходами, не требующими входного символа. Попав в некоторое состояние с помощью ε-перехода, мы должны предполагать, что автомат находится в обоих состояниях. Использование ε-переходов позволяет с легкостью создать автомат на основе обхода в глубину дерева синтаксического разбора регулярного выражения. Такой автомат будет иметь $O(m)$ состояний, где $m$ — длина регулярного выражения. Эмуляция автомата на строке длиной $n$ символов занимает время $O(mn)$, поскольку каждую пару «состояние/префикс» нужно рассмотреть только один раз.
  - Создание *детерминированного* конечного автомата начинается с построения дерева синтаксического разбора регулярного выражения с соблюдением того условия, что каждый лист должен представлять символ алфавита в образце. Распознав префикс текста, мы оказываемся в некотором подмножестве возможных позиций, которое будет соответствовать состоянию конечного автомата. Алгоритм Бржозовского (Brzozowski) создает этот автомат по одному состоянию за раз по мере надобности. Для некоторых регулярных выражений длиной в $m$ символов потребуется $O(2^m)$ состояний в любом реализующем их детерминированном конечном автомате. В качестве примера можно привести выражение $(a + b)^*a(a + b)(a + b)...(a + b)$. Нет никакой возможности избежать этого экспоненциального «взрыва» сложности по памяти. К счастью, эмуляция входной строки занимает линейное время на любом детерминированном конечном автомате, независимо от размера автомата.

### Реализации

Пакет Grail+ на языке C++ предназначен для символьных вычислений над конечными автоматами и регулярными выражениями. Пакет позволяет выполнять преобразования между разными представлениями автомата и минимизировать автоматы. Он может обрабатывать автоматы большого размера, определенные на больших алфавитах. Код и документацию этого пакета можно найти на веб-сайте **http://www.csit.upei.ca/~ccampeanu/Grail/**, где также предоставлены ссылки на многие другие пакеты.

Библиотека OpenFst Library (**http://www.openfst.org/**) предназначена для создания, комбинирования, оптимизирования и исследования взвешенных конечных преобразователей, которые представляют собой обобщение конечных автоматов, чьи выходные символы не обязательно такие же, как и входные. В библиотеке предоставляются возможности минимизации и преобразований к детерминированным автоматам.

Пакет JFKAP содержит графические инструменты для изучения основных понятий теории автоматов. В пакет включены функции для выполнения преобразований между детерминированными конечными автоматами, недетерминированными конечными автоматами и регулярными выражениями, а также функции для минимизации автоматов. Поддерживаются автоматы более высокого уровня, в том числе контекстно-

свободные языки и машины Тьюринга. Пакет JFLAP можно загрузить с веб-сайта **http://www.jflap.org**. Там же доступна книга [RF06].

#### Примечания

В работе [Aho90] приводится хороший обзор алгоритмов поиска по образцу, в том числе и для образцов регулярных выражений. Метод поиска образцов в виде регулярных выражений, использующий ε-переходы, представлен в работе [Tho68]. Описание методов поиска по образцу с помощью конечных автоматов можно найти в книге [AHU74]. Обсуждение конечных автоматов и теории вычислений содержится в книгах [HK11], [HMU06] и [Sip05]. Основным форумом специалистов в этой области является конференция CIAA (Conference on Implementation and Application of Automata — конференция по реализации и применению автоматов).

Оптимальный алгоритм с временем исполнения $O(n \lg n)$ для минимизации количества состояний детерминированного конечного автомата представлен в работе [Hop71]. Метод производных для создания конечного автомата из регулярного выражения описан в работе [Brz64] и изложен более подробно в работе [BS86]. Описание алгоритма Бржозовского вы найдете в том числе и в книге [Con71]. Среди последних работ по инкрементальному созданию и оптимизации автоматов можно назвать статью [Wat03]. Задача преобразования детерминированного конечного автомата в минимальный недетерминированный конечный автомат (см. [JR93]), а также задача проверки на эквивалентность двух недетерминированных конечных автоматов (см. [SM73]) являются PSPACE-полными.

Неэффективность алгоритмических подходов для решения задач поиска образцов, представленных регулярными выражениями, используется для атак DoS (Denial of Service — отказ от обслуживания) на веб-сайты. Если общедоступный код содержит регулярное выражение, требующее вычислений со сверхлинейной временной сложностью (в наихудшем случае), злоумышленник может перегрузить систему, предоставив для обработки «патологический» ввод, синтаксический анализ которого занимает чрезвычайно длительное время. Обсуждение такого типа атак приводится в [DCSL18].

### Родственные задачи

Задача выполнимости (см. *разд. 17.10*), поиск строк (см. *разд. 21.3*).

## 21.8. Максимальная общая подстрока

**Вход.** Множество $S$ строк $S_1, ..., S_n$.

**Задача.** Найти такую максимальную строку $S'$, все символы которой входят в виде подстроки в каждую строку $S_i$, где $1 \leq i \leq n$ (рис. ЦВ-21.9).

### Обсуждение

Задача поиска максимальной общей подстроки, или подпоследовательности, возникает при сравнении текстов. Особенно важным приложением является выяснение степени сходства биологических последовательностей. Гены белков эволюционируют со временем, но их важнейшие участки изменяться не должны. Максимальная общая подпоследовательность вариаций одного гена в разных биологических видах позволяет получить представление о генном материале, не подвергшемся изменениям.

Задача поиска максимальной общей подстроки представляет собой частный случай задачи вычисления расстояния редактирования (см. *разд. 21.4*), в котором замены сим-

волов недопустимы, а разрешаются только вставка и удаление. При таких условиях расстояние редактирования между строками $P$ и $T$ равно $n + m - 2|lcs(P, T)|$, поскольку мы можем удалять из $P$ символы, отсутствующие в $lcs(P, T)$, а затем вставлять символы строки $T$, чтобы преобразовать строку $P$ в строку $T$.

При решении этой задачи возникают следующие вопросы.

- *Требуется ли найти общую подстроку?*

  При проверке текстов на уникальность нам нужно найти самую длинную общую фразу в двух и более документах. Так как фразы являются строками символов, ищется максимальная *подстрока*, общая для всех текстов.

  Максимальную общую подстроку для множества строк можно найти за линейное время, используя суффиксные деревья (см. *разд. 15.3*). Решение заключается в том, чтобы создать суффиксное дерево, содержащее все строки, пометить каждый лист входной строкой, которую он представляет, а потом выполнить обход в глубину и найти самый дальний узел с потомками из каждой входной строки.

- *Требуется ли найти общую подпоследовательность разбросанных символов?*

  Далее в этом разделе мы будем рассматривать только задачу поиска общих подпоследовательностей разбросанных символов. Алгоритм поиска таких подпоследовательностей является частным случаем алгоритма динамического программирования для вычисления расстояния редактирования. Реализация этого алгоритма на языке C приводится в листинге 10.14.

  Пусть $M[i, j]$ — количество символов самой длинной общей подпоследовательности строк $S[1]$, ..., $S[i]$ и $T[1]$, ..., $T[j]$. Когда $S[i] \neq T[j]$, совпадение последней пары символов невозможно, поэтому $M[i, j] = \max(M[i, j-1], M[i-1, j])$. Но если $S[i] = T[j]$, мы также можем выбрать этот символ для нашей подстроки, поэтому $M[i, j] = \max(M[i-1, j-1] + 1, M[i-1, j], M[i, j-1])$.

  Это рекуррентное соотношение вычисляет длину максимальной общей подпоследовательности за время $O(nm)$. Саму общую последовательность можно воссоздать, двигаясь в обратном направлении от $M[n, m]$ и выясняя, какие символы совпали.

- *Как поступать, если строки являются перестановками?*

  Перестановки — это строки, в которых символы не повторяются. Две перестановки задают $n$ пар совпадающих символов, и в этом случае приведенный ранее алгоритм исполняется за время $O(n \lg n)$. Важным частным случаем является поиск максимальной возрастающей подпоследовательности числовой последовательности $p$. Отсортировав последовательность $p$ в возрастающем порядке, мы получим последовательность $s$. Максимальная общая подпоследовательность последовательностей $p$ и $s$ дает нам максимальную возрастающую подпоследовательность.

- *Как поступать, если количество наборов совпадающих символов сравнительно невелико?*

  Для строк, содержащих не слишком много экземпляров одного и того же символа, как в упомянутых ранее перестановках, существует более быстрый алгоритм. Пусть $r$ — количество пар таких позиций $(i, j)$, для которых $S_i = T_j$. Здесь каждая из $r$ пар определяет точку на плоскости.

Полный набор таких точек можно вычислить за время $O(n + m + r)$, используя метод раскладывания по корзинам. Для каждого символа алфавита $c$ и каждой строки ($S$ или $T$) создается корзина, после чего номера позиций каждого символа строки распределяются по соответствующим корзинам. Потом из каждой пары $s \in S_c$ и $t \in T_c$ в корзинах $S_c$ и $T_c$ создается точка $(s, t)$.

Общая подпоследовательность описывает монотонно неубывающий путь по этим точкам — т. е. путь, все перемещения по которому выполняются только вверх и вправо. Самый длинный такой путь можно найти за время $O((n + r)\lg n)$. Точки сортируются в порядке возрастания $x$-координаты, а конфликты с одинаковым значением этой координаты разрешаются в пользу точки с бóльшим значением $y$-координаты. В этом порядке мы вставляем точки и отслеживаем минимальную конечную $y$-координату любого пути, проходящего ровно через $k$ точек, где $1 \le k \le n$. Каждая новая точка $(p_x, p_y)$ изменяет только один из этих путей, либо определяя новую максимальную подпоследовательность, либо уменьшая значение $y$-координаты кратчайшего пути, конечная точка которого находится выше точки $p_y$.

- *Как поступать, если задано несколько строк?*

  Базовый алгоритм динамического программирования можно обобщить для работы с $k$ строками. Время исполнения в этом случае будет равно $O(2^k n^k)$, где $n$ — длина максимальной строки. Алгоритм имеет сложность, экспоненциально зависящую от количества строк $k$, и поэтому он годится только для небольшого количества строк. Эта задача является NP-полной, поэтому не следует надеяться на появление более быстрого точного алгоритма в ближайшее время.

  Для решения задачи выравнивания нескольких последовательностей было предложено большое количество эвристических методов. Часто эти алгоритмы начинают работу с вычисления попарного выравнивания для каждой пары строк. В одном из подходов две наиболее похожие последовательности объединяются в одну и заменяются ею, и этот процесс повторяется до тех пор, пока все последовательности не будут слиты в одну. Проблема здесь в том, что для двух строк часто существует много разных выравниваний оптимальной стоимости. Выбор самого подходящего из них зависит от оставшихся последовательностей, подлежащих слиянию, а они алгоритму неизвестны.

### Реализации

Для выравнивания нескольких последовательностей ДНК существует много программ. В частности, популярным инструментом выравнивания протеиновых последовательностей является программа ClustalW (см. [THG94]). Программу можно загрузить с веб-сайта **https://www.ebi.ac.uk/Tools/msa/**. Еще одно заслуживающее внимания программное средство выравнивания нескольких биологических последовательностей — это пакет MSA, который можно загрузить с веб-страницы **https://www.ncbi.nlm.nih.gov/CBBresearch/Schaffer/msa.html**.

Любую из программ динамического программирования для нечеткого сравнения строк, рассмотренных в *разд. 21.4*, можно использовать для поиска максимальной общей подстроки.

Библиотека Combinatorica (см. [PS03]) содержит реализацию (на языке пакета Mathematica) алгоритма создания максимальной возрастающей подпоследовательности перестановки. Этот алгоритм основан не на динамическом программировании, а на использовании таблиц Янга (подробности см. в *разд. 22.1.8*).

#### Примечания

Обзоры алгоритмов поиска максимальных общих подпоследовательностей можно найти в [BHR00] и [GBY91]. Алгоритм для последовательностей, имеющих мало или вовсе не имеющих совпадений, представлен в работе [HS77], его недавнее улучшение описывается в работе [IR09]. Описание алгоритма приводится также в книгах [Aho90] и [Man89]. В последнее время наблюдается неожиданное повышение интереса к этой задаче, в частности к разработке эффективных алгоритмов поиска максимальной общей подпоследовательности, применяющих параллелизм на уровне битов (см. [CIPR01]). В работе [MP80] представлен быстрый алгоритм поиска максимальной общей последовательности для алфавитов фиксированного размера. Алгоритм основан на методе «четырех русских» и имеет время исполнения, равное $O(mn/\log(\min\{m, n\}))$. Существование алгоритма с временной сложностью меньшей, чем квадратичная, требует опровержения сильной гипотезы об экспоненциальном времени. Подробности см. в [ABW15] и [BK18].

Создадим две случайные строки длиной $n$ на алфавите из $a$ символов. Какова ожидаемая длина максимальной общей подстроки? Эта задача является предметом активных исследований, прекрасный обзор которых приведен в работе [Dan94].

Выравнивание нескольких биологических последовательностей представляет собой отдельную область, хорошим введением в которую послужат книги [Gus97] и [CP18]. Свежий обзор состояния дел вы найдете в работе [Not02]. Сложность задачи выравнивания нескольких последовательностей следует из сложности задачи поиска минимальной общей подстроки для больших наборов строк (см. [Mai78]).

Среди приложений задачи поиска максимальной общей подстроки мы упомянули возможность ее применения для выявления плагиата. Интересные рассуждения о том, как реализовать детектор плагиата для компьютерных программ, приведены в работе [SWA03].

### Родственные задачи

Нечеткое сравнение строк (см. *разд. 21.4*), поиск минимальной общей надстроки (см. *разд. 21.9*).

## 21.9. Поиск минимальной общей надстроки

**Вход.** Множество строк $S = \{S_1, ..., S_m\}$.

**Задача.** Найти минимальную строку $S$, содержащую в виде подстроки каждую строку $S_i$ (рис. 21.10).

### Обсуждение

Задача поиска минимальной общей надстроки, или надпоследовательности, возникает в разных приложениях. Однажды заядлый игрок казино спросил у меня, как восстановить последовательность символов на диске игрового автомата. При очередном раунде игры диск игрового автомата после вращения останавливается в случайной позиции,

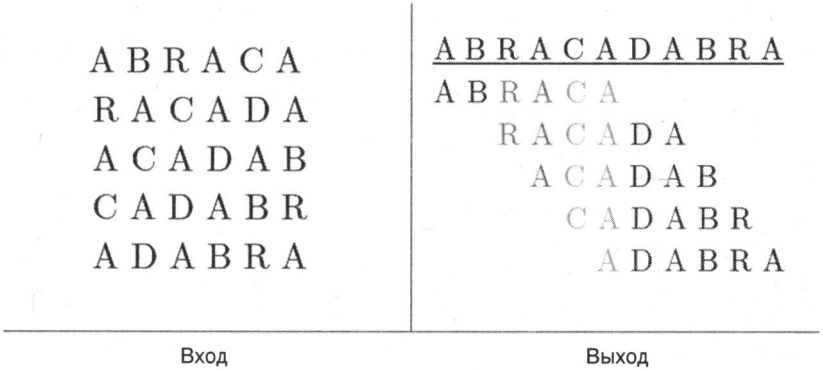



**Рис. 21.10.** Поиск минимальной общей надстроки

показывая выпавший символ, а также непосредственно предшествующий и следующий ему символы. При наблюдении за достаточно большим количеством раундов игры можно выяснить порядок символов каждого диска игрового автомата, решив задачу поиска минимальной общей (циклической) надстроки для трехсимвольных подстрок, полученных в результате наблюдения.

Более важное применение задачи поиска минимальной общей надстроки — сжатие матрицы. Предположим, имеется разреженная матрица $M$ размером $n \times m$ — т. е. значение большинства ее элементов равно нулю. Каждую строку матрицы можно разбить на $m/k$ подстрок из $k$ элементов, а потом создать минимальную общую надстроку $S'$ из этих подстрок. Теперь матрицу можно представить этой надстрокой совместно с массивом указателей размером $n \times m/k$, обозначающих начало каждой последовательности в надстроке. Доступ к любому элементу по-прежнему занимает постоянное время, но если $|S| << mn$, то экономия памяти будет значительной.

Возможно, самым популярным приложением задачи поиска минимальной общей надстроки является сборка ДНК. Роботы без труда собирают последовательности из 100–1000 базовых пар ДНК, но наибольший интерес вызывает секвенирование длинных молекул. В процессе сборки последовательностей методом «дробовика» создается множество копий целевой молекулы, которые разбиваются на случайные фрагменты. Затем выполняется секвенирование этих фрагментов, и их минимальная надстрока считается корректной последовательностью.

Найти надстроку набора строк нетрудно, поскольку мы можем просто конкатенировать исходные строки. Проблема в том, чтобы найти *кратчайшую* надстроку. Задача поиска минимальной общей надстроки является NP-полной для строк всех классов.

Задачу поиска минимальной общей надстроки можно свести к задаче коммивояжера (см. *разд. 19.4*). Для этого создаем ориентированный граф перекрытий $G$, в котором вершина $v_i$ представляет строку $S_i$. Присваиваем ребру $(v_i, v_j)$ вес, равный длине строки $S_i$, уменьшенной на длину перекрытия строк $S_j$ и $S_i$. Например, $w(v_i, v_j) = 1$ для $S_i = abc$ и $S_j = bcd$. Путь с минимальным весом, проходящий через все вершины, определяет минимальную общую надстроку. Веса ребер несимметричны — обратите внимание, что на рис. 21.10 $w(v_j, v_i) = 3$. К сожалению, несимметричные задачи коммивояжера намного сложнее симметричных.

Стандартным решением является аппроксимирование минимальной общей надстроки с помощью «жадного» алгоритма. Находим пару строк с максимальным перекрытием. Заменяем эти строки их объединением и повторяем процесс до тех пор, пока не останется только одна строка. Для этого эвристического алгоритма можно создать реализацию с линейным временем исполнения. По-видимому, самым дорогим этапом является создание графа перекрытий. Поиск максимального перекрытия двух строк длиной $l$ методом исчерпывающего перебора занимает время $O(l^2)$ для каждой из $O(n^2)$ пар строк. Но для решения этой задачи существуют более быстрые методы, основанные на использовании суффиксных деревьев (см. *разд. 15.3*). Создаем дерево, содержащее все суффиксы всех строк множества $S$. Строка $S_i$ перекрывает строку $S_j$ тогда и только тогда, когда суффикс строки $S_i$ совпадает с префиксом строки $S_j$, а это событие отображается вершиной суффиксного дерева. Обход таких вершин в порядке увеличения расстояния от корня определяет соответствующий порядок слияния строк.

Какова эффективность этого «жадного» эвристического алгоритма? Очевидно, существуют случаи, заставляющие его создать надстроку, вдвое превышающую оптимальную. Например, оптимальный порядок слияния строк $c(ab)^k$, $(ba)^k$ и $(ab)^kc$ — слева направо. Но наш «жадный» алгоритм начинает со слияния первой и третьей строки, не оставляя для средней строки возможности перекрытия. Таким образом, надстрока, возвращаемая «жадным» алгоритмом, никогда не будет превышать оптимальную более чем в 3,5 раза, а на практике результат обычно даже лучше.

Задача создания надстрок значительно усложняется, когда вход содержит как положительные строки (каждая из которых должна быть подстрокой надстроки), так и отрицательные строки (каждая из которых не может входить в конечный результат). Задача выявления таких строк является NP-полной, если только не разрешено добавить в алфавит дополнительный символ, чтобы использовать его как разделитель.

### Реализации

Для сборки последовательностей ДНК существует несколько высокопроизводительных программ. Такие программы исправляют ошибки секвенирования, поэтому конечный результат не обязательно будет надстрокой входных строк. Как минимум эти программы могут послужить в качестве моделей, когда вам требуется найти недлинную корректную надстроку.

Самыми последними разработками в этой серии являются программы CAP3 (см. [HM99]) и PCAP (см. [HWA+03]), которые можно загрузить с веб-сайта **http://seq.cs.iastate.edu/**. Они применялись в проектах сборки генов млекопитающих, когда одновременно были задействованы сотни миллионов участков ДНК.

Программа Celera, которая использовалась для секвенирования генома человека, теперь доступна для общего пользования. Ее можно загрузить с веб-страницы **https://sourceforge.net/projects/wgs-assembler/**.

#### *Примечания*

Описание задачи поиска минимальной общей подстроки и ее применения в сборке последовательностей ДНК методом «дробовика» представлено в книге [MKT07] и в работах [Mye99a] и [SP15]. В статье [KM95] приводится алгоритм решения более общего случая задачи поиска минимальной общей надстроки, когда предполагается, что исходные стро-

ки содержат ошибки. Эта статья рекомендуется для чтения всем интересующимся сборкой фрагментов ДНК.

В работе [BJL+94] приведены первые аппроксимирующие алгоритмы поиска минимальной общей надстроки, в которых применяется разновидность «жадного» эвристического метода и которые выдают решения, имеющие постоянный коэффициент. В результате более поздних исследований (см. [Muc13] и [Pal14]) удалось уменьшить этот коэффициент до 2,367, что является шагом вперед на пути к ожидаемому коэффициенту 2. В настоящее время самое лучшее приблизительное решение, выдаваемое стандартным «жадным» эвристическим алгоритмом, имеет коэффициент 3,5 (см. [KS05a]). Эффективные реализации такого алгоритма описаны в работе [Gus94].

Отчет об экспериментах с эвристическими методами поиска минимальных общих надстрок представлен в статье [RBT04]. На основе этих экспериментов можно предположить, что для входных экземпляров разумного размера «жадные» эвристические методы обычно выдают решения, имеющие коэффициент лучший чем 1,4%. Результаты экспериментов с использованием подходов, основанных на генетических алгоритмах, приведены в статье [ZS04]. В работе [YZ99] содержатся аналитические результаты, демонстрирующие очень небольшое сокращение минимальной общей надстроки случайных последовательностей, обусловленное тем, что ожидаемая длина перекрытия двух произвольных строк невелика.

### Родственные задачи

Суффиксные деревья (см. *разд. 15.3*), сжатие текста (см. *разд. 21.5*).

# Ресурсы

Эта глава содержит краткое описание ресурсов, с которыми должен быть знаком каждый разработчик алгоритмов. Хотя часть этой информации уже была представлена в других местах каталога задач, здесь собраны наиболее важные ссылки.

## 22.1. Программные системы

Хороший разработчик алгоритмов не изобретает колесо, а хороший программист не создает код, который уже был создан другими. Лучше всего на эту тему выразился Пикассо: «Хорошие художники заимствуют. Великие художники крадут».

Здесь необходимо сказать несколько слов по поводу воровства. Многие из программных продуктов, описываемых в этой книге, доступны только для исследовательских или образовательных целей. Для использования таких продуктов в коммерческих целях необходимо заключить лицензионное соглашение с их авторами. Я настоятельно рекомендую вам соблюдать это правило. Лицензионные условия большинства академических организаций обычно вполне скромны. Факт использования кода в коммерческих разработках часто бывает важнее для его автора, чем финансовая сторона. Поступайте честно и приобретайте необходимые лицензии. Условия лицензии и контактные данные обычно указаны в документации на программное обеспечение или доступны на веб-сайте разработчика.

Хотя многие из описанных здесь программ можно получить из нашего хранилища алгоритмов (**www.algorist.com**), я настоятельно рекомендую загружать эти программы с сайтов их разработчиков. Во-первых, существует *очень* большая вероятность, что версия на сайте разработчика является самой свежей. Во-вторых, на сайте разработчика часто можно найти дополнительные вспомогательные файлы и документацию, которые могут оказаться полезными. Наконец, многие разработчики следят за количеством загрузок своих программ, поэтому вы лишите их морального поощрения, если будете загружать их программы с других сайтов.

### 22.1.1. Библиотека LEDA

Библиотека LEDA (Library of Efficient Data Types and Algorithms — библиотека эффективных типов данных и алгоритмов) — это, пожалуй, самый лучший из доступных ресурсов с реализациями комбинаторных методов. Библиотека создана группой разработчиков из Института Макса Планка в городе Саарбрюкен, Германия. Библиотека LEDA уникальна благодаря высочайшей квалификации ее создателей и большому количеству средств, вложенных в этот проект.

Эта библиотека предлагает обширную коллекцию хорошо реализованных на языке C++ структур данных и типов. Особенно полезным является тип graph, который поддерживает все основные операции. В библиотеку входит набор программ для работы с графами, наглядно демонстрирующих, как можно аккуратно и кратко реализовать алгоритмы, используя типы из библиотеки LEDA. В библиотеке имеются хорошие реализации словарей и очередей с приоритетами. Дополнительную информацию о библиотеке вы найдете в книге [MN99].

Библиотеку можно получить исключительно через компанию Algorithmic Solutions Software GmbH (**https://www.algorithmic-solutions.com/**). Таким образом обеспечивается профессиональная поддержка библиотеки и гарантируется регулярный выпуск новых версий. Бесплатная версия библиотеки содержит все основные структуры данных, включая словари, очереди с приоритетами, графы и числовые типы. В ней отсутствуют лишь исходный код и некоторые особо сложные алгоритмы. Но плата за лицензию на использование полной версии библиотеки невелика, к тому же у пользователя имеется возможность бесплатно загрузить пробную версию полной библиотеки. Так что посетите веб-сайт библиотеки и посмотрите, что в ней может быть полезного для вас.

## 22.1.2. Библиотека CGAL

Библиотека CGAL (Computational Geometry Algorithms Library, библиотека алгоритмов вычислительной геометрии) содержит эффективные и надежные реализации алгоритмов вычислительной геометрии на языке C++. Библиотека весьма обширна и включает широкий спектр реализаций алгоритмов создания триангуляционных разбиений, диаграмм Вороного, конфигураций прямых, альфа-очертаний, выпуклых оболочек, а также структуры данных для геометрического поиска. Предлагаемые реализации предназначены для работы в двух- и трехмерном пространстве, а некоторые — и в пространствах с бóльшим количеством измерений.

Сайт библиотеки CGAL (**www.cgal.org**) — это первое место, где вы должны искать профессиональные программы для геометрических вычислений. Библиотека CGAL распространяется по схеме двойного лицензирования. Ее можно бесплатно использовать вместе с программным обеспечением с открытым исходным кодом, но для использования в других ситуациях необходимо приобрести коммерческую лицензию.

## 22.1.3. Библиотека Boost

Библиотека Boost (**www.boost.org**) содержит популярную коллекцию прошедших экспертную оценку бесплатных переносимых исходных кодов библиотек на языке C++. Условия лицензии библиотеки предусматривают как коммерческое, так и некоммерческое ее использование.

Библиотека Boost Graph Library, пожалуй, лучше всего подходит для читателей этой книги. Руководство пользователя можно найти в [SLL02] и на странице **http://www.boost.org/libs/graph/doc**. Библиотека включает реализации списков смежности, матриц смежности и списков ребер, а также неплохую коллекцию базовых алгоритмов для работы с графами. Интерфейс и компоненты библиотеки являются обобщенными в том

смысле, в каком этот термин употребляется в стандартной библиотеке шаблонов STL языка C++. Другие интересные коллекции из библиотеки Boost содержат программы обработки строк и пакеты для математических вычислений.

## 22.1.4. Библиотека Netlib

Библиотека Netlib (**https://netlib.org**) — это хранилище математического программного обеспечения, содержащее большой объем кода, а также множество таблиц и статей. Важные достоинства библиотеки Netlib — широкий спектр ее ресурсов и легкость доступа к ним. Если вам потребуется специализированная математическая программа, начните ее поиски с библиотеки Netlib.

Служба GAMS (Guide to Available Mathematical Software — путеводитель по существующему математическому программному обеспечению) индексирует библиотеку Netlib и другие хранилища математического программного обеспечения. Она поддерживается Национальным институтом стандартов и технологий США (NIST) и доступна по адресу **http://gams.nist.gov**.

## 22.1.5. Коллекция алгоритмов ассоциации ACM

Одним из первых механизмов распространения реализаций полезных алгоритмов стала коллекция CALGO (Collected Algorithms of the ACM — сборник алгоритмов ассоциации ACM). Впервые коллекция была представлена в журнале «Communications of the ACM» в 1960 году, и тогда в нее вошли такие знаменитые алгоритмы, как алгоритм Флойда для создания пирамиды с линейным временем исполнения. В настоящее время коллекция сопровождается журналом «ACM Transactions on Mathematical Software». Для каждого алгоритма и его реализации приводится краткое описание в статье журнала, а реализация проверяется и добавляется в коллекцию. Программы доступны по адресу **http://www.acm.org/calgo** и на веб-сайте библиотеки Netlib.

На сегодняшний день в коллекции имеется почти 1000 алгоритмов. Большинство программ написаны на языке FORTRAN и относятся к математическим вычислениям, хотя в ней нашли свое место и несколько интересных комбинаторных алгоритмов. Поскольку реализации прошли экспертную оценку, они считаются более надежными, чем аналогичные программы из других источников.

## 22.1.6. Сайты GitHub и SourceForge

Платформа GitHub (**https://github.com**) для разработчиков программного обеспечения содержит свыше 28 миллионов общедоступных репозиториев и является наибольшим в мире хранилищем исходного кода. Посещение этого хранилища должно быть вторым этапом поиска реализаций алгоритмов. Направляйтесь туда сразу же после ознакомления с очередным алгоритмом, приведенным в каталоге этой книги. Там вы сможете с большой вероятностью найти все интересные новые коды, включая свыше десяти упоминаемых в каталоге систем.

Сайт SourceForge (**http://sourceforge.net**) — это более старый ресурс программного обеспечения с открытым исходным кодом, насчитывающий свыше 160 тысяч зарегистрированных проектов. Большинство из этих проектов представляет интерес лишь для

узкого круга потребителей, но вы тоже сможете найти там много полезного материала, в частности, библиотеки JUNG и JGraphT для работы с графами и средства оптимизации lpsolve и JGAP.

## 22.1.7. Система Stanford GraphBase

Система Stanford GraphBase интересна во многих отношениях. Прежде всего, она создана в соответствии с принципом «грамотного программирования», а это означает, что ее можно читать. И если чьи-либо программы заслуживают, чтобы их читали, так это программы Дональда Кнута. Его книга [Knu94] описывает полный исходный код системы Stanford GraphBase.

Система GraphBase содержит реализации многих важных комбинаторных алгоритмов, включая алгоритмы поиска паросочетаний, вычисления минимальных остовных деревьев и построения диаграмм Вороного, а также специализированные средства для создания графов-расширителей и создания комбинаторных объектов. Кроме того, система содержит программы для решения многих развлекательных задач, включая создание «лесенок слов» (цепочек, в которых каждое слово отличается от предыдущего на одну букву) и определения доминирующих отношений среди футбольных команд. Домашняя страница системы находится в Интернете по адресу **http://www-cs-faculty.stanford.edu/~knuth/sgb.html**.

С системой GraphBase интересно экспериментировать, но она мало пригодна для создания приложений общего характера. Ее можно применять самое большее как генератор экземпляров графов, которые могут послужить в качестве тестовых данных для других программ.

## 22.1.8. Пакет Combinatorica

Пакет Combinatorica (см. [PS03]) содержит реализации свыше 450 алгоритмов по комбинаторике и теории графов и широко используется как для исследовательских, так и для образовательных целей.

Хотя (с моей точки зрения) пакет Combinatorica более полный и лучше интегрирован, чем другие библиотеки комбинаторных алгоритмов, он работает очень медленно. так что он лучше всего подходит для решения небольших задач, а также может послужить источником кратких описаний алгоритмов, пригодных для перевода на другие языки программирования (если вы сумеете разобраться в коде Mathematica).

Пакет Combinatorica и связанные с ним ресурсы доступны на веб-сайте **http://www.combinatorica.com**. Вследствие возросшей важности графов большая часть возможностей пакета включена в сам язык Mathematica. Пакет также входит в состав стандартного дистрибутива Mathematica и находится в папке Packages/DiscreteMath/Combinatorica.m.

## 22.1.9. Программы из книг

Многие книги, посвященные алгоритмам, содержат работоспособные реализации алгоритмов, написанные на распространенных языках программирования. Хотя эти реали-

зации предназначены главным образом для иллюстрации материала книги, их вполне можно использовать на практике. Поскольку они, как правило, аккуратно написаны и имеют небольшой размер, то вполне могут послужить хорошей основой для простых приложений.

Далее приводятся описания наиболее интересных программ этого типа. Большинство из них доступны в хранилище алгоритмов по адресу **www.algorist.com**.

### Книга «Programming Challenges»

Если вам нравятся программы на языке C, приведенные в первой части этой книги, вас, вероятно, заинтересуют программы, которые я написал для своей книги [SR03]. Возможно, наибольшую пользу принесут дополнительные примеры алгоритмов динамического программирования, процедуры вычислительной геометрии, такие как создание выпуклой оболочки, а также пакет bignum для выполнения арифметических операций с большими целыми числами. Эта библиотека алгоритмов доступна на сайтах: **https://www.cs.stonybrook.edu/~skiena/392/programs/** и **https://github.com/SkienaBooks/Algorithm-Design-Manual-Programs.**

### Книга «Combinatorial Algorithms for Computers and Calculators»

Эта книга (см. [NW78]) посвящена алгоритмам создания элементарных комбинаторных объектов — таких как перестановки, подмножества и разбиения. У этих алгоритмов, как правило, очень короткие описания, но их трудно найти, и в них не всегда легко разобраться. Для всех алгоритмов в книге имеются реализации на языке FORTRAN, сопровождаемые обсуждением теоретических основ. В ней приводятся алгоритмы как для случайного, так и для последовательного генерирования объектов.

Описания недавно появившихся алгоритмов решения некоторых задач (без предоставления кода программ) можно найти в книге [Wil89]. Реализации этих алгоритмов можно загрузить из нашего хранилища алгоритмов. Мне удалось обнаружить их у Нила Слоуна (Neil Sloane), который хранил эти реализации на магнитной ленте, в то время как у самих авторов их не было!

### Книга «Computational Geometry in C»

Эта книга (см. [O'R01]) является, пожалуй, самым лучшим введением в вычислительную геометрию, благодаря присутствию в ней аккуратно написанных на языке C реализаций основных алгоритмов. Она содержит реализации всех алгоритмов для решения элементарных задач вычислительной геометрии, построения выпуклых оболочек, триангуляций и диаграмм Вороного, а также планирования перемещений. Хотя эти реализации в основном предназначены для иллюстрации материала книги, а не коммерческого использования, они, скорее всего, достаточно надежны. Программы можно загрузить с веб-сайта **http://maven.smith.edu/~orourke/code.html**.

### Книга «Algorithms in C++»

Эта книга (см. [Sed98] и [SW11]) издана в нескольких вариантах, ориентированных на разные языки программирования, включая языки C, C++ и Java. От других книг ее отличает широкий диапазон рассматриваемых тем, в том числе алгоритмов для работы

с числовыми, строковыми и геометрическими объектами. Код на языке Java можно загрузить с веб-сайта **https://algs4.cs.princeton.edu/**.

### Книга «Discrete Optimization Algorithms in Pascal»

Эта книга (см. **http://www.cs.princeton.edu/~rs/**) содержит коллекцию из 28 программ для решения задач дискретной оптимизации. Сюда входят программы для решения задач целочисленного и линейного программирования, задач о рюкзаке, о покрытии множества, а также задач коммивояжера, вершинной раскраски, календарного планирования и оптимизации сетей. Программы доступны по адресу **www.algorist.com**.

## 22.2. Источники данных

При тестировании алгоритмов часто возникает необходимость использовать в качестве входа нетривиальные данные, позволяющие проверить правильность работы алгоритма или сравнить скорость работы разных алгоритмов. Поиск хороших тестовых данных может оказаться трудной задачей. Вот некоторые источники:

- *Проект SNAP (Stanford Network Analysis Project).*

  Этот ресурс совмещает библиотеку общего назначения для сетевого анализа сотен миллионов узлов с хорошей коллекцией графов, созданных на основе реальных данных из социальных сетей, сетей цитирования научных работ и сетей связи. Все эти материалы доступны по адресу **https://snap.stanford.edu/.**

- *Библиотека TSPLIB.*

  Эта широко известная библиотека содержит стандартную тестовую коллекцию сложных экземпляров задачи коммивояжера (см. [Rei91]). Имеющиеся в ней экземпляры задачи представляют собой большие графы, полученные из реальных приложений, таких как проектирование печатных плат и сетей. Библиотека доступна по адресу **http://www.math.uwaterloo.ca/tsp/data**.

- *Stanford GraphBase.*

  Этот пакет программ, написанных Дональдом Кнутом (см. *разд. 22.1.7*), содержит машинно-независимые генераторы разнообразных графов, включая графы, построенные на основе матриц расстояний и произведений искусства и литературы, а также графы, представляющие чисто теоретический интерес.

## 22.3. Библиографические ресурсы

Интернет предоставляет неограниченные возможности людям, интересующимся алгоритмами. Упомяну ресурсы, которыми я пользуюсь чаще всего:

- *Google Scholar.*

  Этот бесплатный ресурс (**http://scholar.google.com**) ограничивается поиском среди научных работ, что позволяет получить более качественные результаты, чем при универсальном поиске. Особенно полезной является возможность выяснить, в каких работах упоминается искомая. Таким образом вы можете обновить свой старый

справочный материал и увидеть, что нового произошло в той или иной области, а также определить научную ценность конкретной статьи.

- *Цифровая библиотека ACM.*

  Эта коллекция библиографических ссылок содержит ссылки на практически все когда-либо опубликованные технические доклады по теории вычислительных систем. Коллекция доступна по адресу **http://portal.acm.org**.

- *Сервер сигнальных экземпляров Arxiv.*

  Содержит свыше 1,6 миллиона работ, в которых ученые приводят результаты своих исследований перед формальной публикацией, ко времени которой они уже устареют. Здесь можно найти последние результаты исследований на любую тему, рассматриваемую в этой книге. Дополнительную информацию см. на веб-сайте **https://arxiv.org/**.

## 22.4. Профессиональные консалтинговые услуги

Консалтинговая фирма Algorist Technologies (**http://www.algorist.com**) предоставляет своим клиентам краткосрочные экспертные услуги по разработке и реализации алгоритмов. Обычно консультант компании выезжает на объект клиента для интенсивного обсуждения задачи с местным коллективом разработчиков в течение срока от одного до трех дней. Компанией Algorist Technologies накоплен впечатляющий список удачных решений по повышению производительности во многих компаниях в различных сферах деятельности.

Для получения дополнительной информации относительно предоставляемых нами услуг пишите по адресу электронной почты **info@algorist.com**.

Algorist Technologies
215 West 92nd St. Suite 1F
New York, NY 10025

# Список литературы


[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gartner. A novel type of skeleton for polygons. J. Universal Computer Science, 1:752–761, 1995.

[AAM18] M. Abrahamsen, A. Adamaszek, and T. Miltzown. The art gallery problem is $\exists r$-complete. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 65–73. ACM, 2018.

[Aar13] Scott Aaronson. *Quantum Computing since Democritus*. Cambridge University Press, 2013.

[Aar15] Scott Aaronson. Quantum machine learning algorithms: Read the fine print. *Nature Physics*, 11(4):291, 2015.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[AB17] Mikhail J Atallah and Marina Blanton. *Algorithms and Theory of Computation Handbook, Volume 1: General Concepts and Techniques*. Chapman & Hall/CRC, 2017.

[ABCC07] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[ABF05] L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 34:1–34:27. Chapman & Hall/CRC Press, 2005.

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 59–78. IEEE, 2015.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[ACFC+16] Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and Shan Muthukrishnan. Forty years of suffix trees. *Communications of the ACM*, 59(4):66–73, 2016.

[ACH+91] E. M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Trans./ PAMI*, 13(3):209–216, 1991.

[ACK01a] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. 6th ACM Symp. on Solid Modeling*, pages 249–260, 2001.

[ACK01b] N. Amenta, S. Choi, and R. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19:127–153, 2001.

[ACL12] Eric Angel, Romain Campigotto, and Christian Laforest. Implementation and comparison of heuristics for the vertex cover problem on huge graphs. In *International Symposium on Experimental Algorithms*, pages 39–50. Springer, 2012.

[ACLZ11] Deepak Ajwani, Adan Cosgaya-Lozano, and Norbert Zeh. Engineering a topological sorting algorithm for massive graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 139–150, 2011.

[ACP+07] H. Ahn, O. Cheong, C. Park, C. Shin, and A. Vigneron. Maximizing the overlap of two planar convex sets under rigid motions. *Computational Geometry: Theory and Applications*, 37:3–15, 2007.

[ADGM07] Lyudmil Aleksandrov, Hristo Djidjev, Hua Guo, and Anil Maheshwari. Partitioning planar graphs with costs and weights. *Journal of Experimental Algorithmics (JEA)*, 11:1–5, 2007.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics, Doklady*, 11:1209–1210, 1970.

[Adl94] L. M. Adleman. Molecular computations of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[AE04] G. Andrews and K. Eriksson. *Integer Partitions*. Cambridge University Press, 2004.

[AF96] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Applied Math.*, 65:21–46, 1996.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium onDiscrete Algorithms*, pages 782–793, 2010.

[AFH02] P. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[AG00] H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 121–153. Elsevier, 2000.

[Aga18] P. Agarwal. Range searching. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1057–1093. CRC Press, 2018.

[AGR01] N. Amato, M. Goodrich, and E. Ramos. A randomized algorithm for triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 26(2):245–265, 2001.

[AGSS89] A. Aggarwal, L. Guibas, J. Saxe, and P. Shor. A linear-time algorithmfor computing the Voronoi diagram of a convex polygon. *Discrete &Computational Geometry*, 4:591–604, 1989.

[AGU72] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing*, 1:131–137, 1972.

[AHK+08] E. Arkin, G. Hart, J. Kim, I. Kostitsyna, J. Mitchell, G. Sabhnani, and S. Skiena. The embroidery problem. In *20th Canadian Conf. Computational Geometry (CCCG)*, 2008.

[Aho90] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 255–300. MIT Press, 1990.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[AI06] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

[AI08] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117, 2008.

[Aig88] M. Aigner. *Combinatorial Search*. Wiley-Teubner, 1988.

[AIR18] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest-neighbor search in high dimensions. *arXiv preprint arXiv:1806.09823*, 2018.

[AITT00] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *J. Algorithms*, 34:203–221, 2000.

[AJNP18] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. *arXiv preprint arXiv:1805.08612*, 2018.

[AK89] E. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, 1989.

[AKL13] Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific Publishing Company, 2013.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[AL97] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[AM93] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 271–280, 1993.

[AMN+98] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998.

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.

[And98] G. Andrews. *The Theory of Partitions*. Cambridge Univ. Press, 1998.

[And05] A. Andersson. Searching and priority queues in $o(\log n)$ time. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 39:1–39:14. Chapman & Hall/CRC Press, 2005.

[ANN+18] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. Hölder homeomorphisms and approximate nearest neighbors. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 159–169. IEEE, 2018.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 574–583. ACM, 2014.

[AP72] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[AP14] Pankaj K Agarwal and Jiangwei Pan. Near-linear algorithms for geometric hitting sets and set covers. In *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, page 271. ACM, 2014.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Info. Proc. Letters*, 8:121–123, 1979.

[Aro98] S. Arora. Polynomial time approximations schemes for Euclidean TSP and other geometric problems. *J. ACM*, 45:753–782, 1998.

[AS00] P. Agarwal and M. Sharir. Arrangements. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier, 2000.

[Ata83] M. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Info. Proc. Letters*, 8:207–209, 1983.

[B+16] Albert-László Barabási et al. *Network Science*. Cambridge University Press, 2016.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[Bap10] Ravindra B Bapat. *Graphs and Matrices*, volume 27. Springer, 2010.

[Bar03] A. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003.

[BB14] Charles H Bennett and Gilles Brassard. Quantum cryptography: public key distribution and coin tossing. *Theor. Comput. Sci.*, 560(12):7–11, 2014.

[BBF99] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM J. Discrete Math.*, 12:289–297, 1999.

[BBPP99] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A sup., pages 1–74. Kluwer, 1999.

[BC19] Stephan Beyer and Markus Chimani. Strong steiner tree approximations in practice. *Journal of Experimental Algorithmics (JEA)*, 24(1):1–7, 2019.

[BCGM13] Gunnar Brinkmann, Kris Coolsaet, Jan Goedgebeur, and Hadrien Mélot. House of graphs: a database of interesting graphs. *Discrete Applied Mathematics*, 161(1-2):311–314, 2013.

[BCGR92] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The SetPlayer system for symbolic computation on power sets. *J. Symbolic Computation*, 14:645–662, 1992.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, 1990.

[BD99] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Disc. Math.*, 201:81–88, 1999.

[BD11] Joseph Blitzstein and Persi Diaconis. A sequential importance sampling algorithm for generating random graphs with prescribed degrees. *Internet Mathematics*, 6(4):489–522, 2011.

[BDH97] C. Barber, D. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans./ on Mathematical Software*, 22:469–483, 1997.

[BDN01] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering (WAE)*, 2001.

[BDY06] K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Trans./ Visualization and Computer Graphics*, 12:773–780, 2006.

[BEB12] Tyson Brochu, Essex Edwards, and Robert Bridson. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics (TOG)*, 31(4):96, 2012.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben90] J. Bentley. *More Programming Pearls*. Addison-Wesley, 1990.

[Ben92a] J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Computing*, 4:387–411, 1992.

[Ben92b] J. Bentley. Software exploratorium: The trouble with qsort. *UNIX Review*, 10(2):85–93, February 1992.

[Ben99] J. Bentley. *Programming Pearls*. Addison-Wesley, second edition, 1999.

[Ber82] D. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, 1982.

[Ber89] C. Berge. *Hypergraphs*. North-Holland, 1989.

[Ber02] M. Bern. Adaptive mesh generation. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, pages 1–56. Springer-Verlag, 2002.

[Ber04] D. Bernstein. Fast multiplication and its applications. http://cr.yp.to/arith.html, 2004.

[Ber15] D. Bertsekas. *Convex Optimization Algorithms*. Athena Scientific Belmont, 2015.

[Ber19] Chris Bernhardt. *Quantum Computing for Everyone*. MIT Press, 2019.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BF00] M. Bender and M. Farach. The LCA problem revisited. In *Proc. 4th Latin American Symp. on Theoretical Informatics*, pages 88–94. Springer-Verlag LNCS vol. 1776, 2000.

[BFH+18] Alon Baram, Efi Fogel, Dan Halperin, Michael Hemmer, and Sebastian Morr. Exact minkowski sums of polygons with holes. *Computational Geometry*, 73:46–56, 2018.

[BFP+72] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[BFS12] G. Blelloch, J. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 308–317. ACM, 2012.

[BFV07] G. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. of Experimental Algorithmics*, 12, 2007.

[BG95] J. Berry and M. Goldberg. Path optimization and near-greedy analysis for graph partitioning: An empirical study. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 1995.

[BGJ+12] Arnab Bhattacharyya, Elena Grigorescu, Kyomin Jung, Sofya Raskhodnikova, and David P Woodruff. Transitive-closure spanners. *SIAM Journal on Computing*, 41(6):1380–1425, 2012.

[BGS95] M Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and nonapproximability—towards tight results. In *Proc. IEEE 36th Symp. Foundations of Computer Science*, pages 422–431, 1995.

[BH90] F. Buckley and F. Harary. *Distances in Graphs*. Addison-Wesley, 1990.

[BH01] G. Barequet and S. Har-Peled. Efficiently approximating the minimum volume bounding box of a point set in three dimensions. *J. Algorithms*, 38:91–109, 2001.

[BHK04] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages, and Programming*, pages 222–233. Springer, 2004.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. String Processing and Information Retreival (SPIRE)*, pages 39–48, 2000.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

[BI15] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proc. Forty-Seventh ACM Symposium on Theory of Computing*, pages 51–58. ACM, 2015.

[BIK+04] H. Bronnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321:25–40, 2004.

[Bis06] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BJL+94] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis. Linear approximation of shortest superstrings. *J. ACM*, 41:630–647, 1994.

[BJL06] C. Buchheim, M. Jünger, and S. Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36:651–665, 2006.

[BJLM83] J. Bentley, D. Johnson, F. Leighton, and C. McGeoch. An experimental study of bin packing. In *Proc. 21st Allerton Conf. on Communication, Control, and Computing*, pages 51–60, 1983.

[BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans./ Pattern Analysis and Machine Intelligence (PAMI)*, 26:1124–1137, 2004.

[BK18] Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1216–1235, 2018.

[BKRV00] A. Blum, G. Konjevod, R. Ravi, and S. Vempala. Semi-definite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, 235:25–42, 2000.

[BL30] Edward Bulwer Lytton. *Paul Clifford*. Henry Colburn and Richard Bentley, London, 1830.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *J. Computer System Sciences*, 13:335–379, 1976.

[BLS91] D. Bailey, K. Lee, and H. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:357–371, 1991.

[BLT12] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, pages 1177–1184. ACM, 2012.

[Blu67] H. Blum. A transformation for extracting new descriptions of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, 1967.

[BLW76] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory 1736–1936*. sndon Press, 1976.

[BM53] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*. Macmillian, 1953.

[BM77] R. Boyer and J. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BM01] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*, pages 245–250, 2001.

[BM05] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2005.

[BMRS16] Kevin Buchin, Wouter Meulemans, André Van Renssen, and Bettina Speckmann. Area-preserving simplification and schematization of polygonal subdivisions. *ACM Transactions on Spatial Algorithms and Systems*, 2(1):2, 2016.

[BMS+16] Aydın Bulu, c, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[BMSW13] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society, 2013.

[BN09] Albert Boggess and Francis J. Narcowich. *A First Course in Wavelets with Fourier Analysis*. John Wiley & Sons, second edition, 2009.

[BO79] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, 1979.

[BO83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. Fifteenth ACM Symp. on Theory of Computing*, pages 80–86, 1983.

[Bol01] B. Bollobas. *Random Graphs*. Cambridge Univ. Press, second edition,2001.

[Bot12] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[BP76] E. Balas and M. Padberg. Set partitioning — a survey. *SIAM Review*, 18:710–760, 1976.

[BR95] A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, 1995.

[Bra99] R. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, third edition, 1999.

[Bra08] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

[Brè79] D. Brèlaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[Bri88] E. Brigham. *The Fast Fourier Transform*. Prentice, facimile edition, 1988.

[Bro95] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, twentieth anniversary edition, 1995.

[Bro97] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997*, pages 21–29. IEEE, 1997.

[BRS+10] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Stefan D. Leigh, M. Levenson, M. Vangel, Nathanael A. Heckert, and D. L. Banks. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, NIST, 2010.

[Bru07] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, fifth edition, 2007.

[Brz64] J. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[BS76] J. Bentley and M. Shamos. Divide-and-conquer in higher-dimensional space. In *Proc. Eighth ACM Symp. Theory of Computing*, pages 220–230, 1976.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory: Efficient Algorithms*, volume 1. MIT Press, 1996.

[BS97] R. Bradley and S. Skiena. Fabricating arrays of strings. In *Proc. First Int. Conf. Computational Molecular Biology (RECOMB '97)*, pages 57–66, 1997.

[BS07] A. Barvinok and A. Samorodnitsky. Random weighting, asymptotic counting and inverse isoperimetry. *Israel Journal of Mathematics*, 158:159–191, 2007.

[BS13] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[BSA18] M. Bern, J. Shewchuk, and N. Amenta. Triangulations and mesh generation. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 763–786. CRC Press, 2018.

[BT92] J. Buchanan and P. Turner. *Numerical Methods and Analysis*. McGraw-Hill, 1992.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, volume 24. Springer Science & Business Media, 2011.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms*. Addison-Wesley, third edition, 1999.

[BW91] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 3:225–242, 1991.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BW00] R. Borndorfer and R. Weismantel. Set packing relaxations of some integer programs. *Math. Programming A*, 88:425–450, 2000.

[BZ10] Richard P. Brent and Paul Zimmermann. *Modern computer Arithmetic*, volume 18. Cambridge University Press, 2010.

[Can87] J. Canny. *The complexity of robot motion planning*. MIT Press, 1987.

[Cas95] G. Cash. A fast computer algorithm for finding the permanent of adjacency matrices. *J. Mathematical Chemistry*, 18:115–119, 1995.

[CB04] C. Cong and D. Bader. The Euler tour technique and parallel rooted spanning tree. In *Int. Conf. Parallel Processing (ICPP)*, pages 448–457, 2004.

[CC92] S. Carlsson and J. Chen. The complexity of heaps. In *Proc. Third ACM-SIAM Symp. on Discrete Algorithms*, pages 393–402, 1992.

[CC97] W. Cook and W. Cunningham. *Combinatorial Optimization*. John Wiley & Sons, 1997.

[CC16] S. Chapra and R. Canale. *Numerical Methods for Engineers*. McGraw-Hill, seventh edition, 2016.

[CCDG82] P. Chinn, J. Chvátolvá, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices — a survey. *J. Graph Theory*, 6:223–254, 1982.

[CCL15] Yixin Cao, Jianer Chen, and Yang Liu. On feedback vertex set: new measure and new structures. *Algorithmica*, 73(1):63–86, 2015.

[CD85] B. Chazelle and D. Dobkin. Optimal convex decompositions. In G. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, 1985.

[CDG+18] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucky, and Michael Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 979–990. IEEE, 2018.

[CDL86] B. Chazelle, R. Drysdale, and D. Lee. Computing the largest empty rectangle. *SIAM J. Computing*, 15:300–315, 1986.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments. *J. ACM*, 39:1–54, 1992.

[CFT99] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999.

[CFT00] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[CG94] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical Report 94–1523, Department of Computer Science, Stanford University, 1994.

[CGJ98] C. R. Coullard, A. B. Gamble, and P. C. Jones. Matching problems in selective assembly operations. *Annals of Operations Research*, 76:95–107, 1998.

[CGJ+13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2013.

[CGK+97] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proc. Symp. on Discrete Algorithms (SODA)*, pages 324–333, 1997.

[CGL85] B. Chazelle, L. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CGM+98] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push: a computational study of bipartite matching and unitcapacity flow algorithms. *J. Experimental Algorithmics*, 3, 1998.

[CGR99] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Prog.*, 10:129–174, 1999.

[CGS99] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Computing*, 28:1326–1346, 1999.

[CH06] D. Cook and L. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[CH09] Graham Cormode and Marios Hadjieleftheriou. Finding the frequent items in streams of data. *Communications of the ACM*, 52(10):97–105, 2009.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackerman type complexity. *J. ACM*, 47:1028–1047, 2000.

[Che85] L. P. Chew. Planing the shortest path for a disc in $O(n^2 \lg n)$ time. In *Proc. First ACM Symp. Computational Geometry*, pages 214–220, 1985.

[Che15] Chi-hau Chen. *Handbook of Pattern Recognition and Computer Vision*. World Scientific, 2015.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1976.

[Chu97] F. Chung. *Spectral Graph Theory*. AMS, 1997.

[Chv83] V. Chvatal. *Linear Programming*. Freeman, 1983.

[CIPR01] M. Crochemore, C. Iliopolous, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Info. Processing Letters*, 80:279–285, 2001.

[CJCG+13] Edward G. Coffman Jr, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: survey and classification. *Handbook of Combinatorial Optimization*, pages 455–531, 2013.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-Quantum Cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.

[CK94] A. Chetverin and F. Kramer. Oligonucleotide arrays: New concepts and possibilities. *Bio/Technology*, 12:1093–1099, 1994.

[CK97] Pierluigi Crescenzi and Viggo Kann. Approximation on the web: A compendium of np optimization problems. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 111–118. Springer, 1997.

[CK05] Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005.

[CK12] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing*. Centage Learning, seventh edition, 2012.

[CKPT17] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review*, 24:63–79, 2017.

[CL98] M. Crochemore and T. Lecroq. Text data compression algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 12.1–12.23. CRC Press, 1998.

[CL13] Ángel Corberán and Gilbert Laporte. *Arc Routing: Problems, Methods, and Applications*. SIAM, 2013.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395, Pittsburgh, PA, 1992.

[CLP11] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry*, pages 1–10. ACM, 2011.

[CLRS09] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[CM99] G. Del Corso and G. Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62:189–203, 1999.

[CM13] Sergio Cabello and Bojan Mohar. Adding one edge to planar graphs makes crossing number and 1-planarity hard. *SIAM Journal on Computing*, 42(5):1803–1829, 2013.

[CN18] Timothy M. Chan and Yakov Nekrich. Towards an optimal method for dynamic planar point location. *SIAM Journal on Computing*, 47(6):2337–2361, 2018.

[CNAO85] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *Journal of Computer and System Sciences*, 30(1):54–76, 1985.

[Coh94] E. Cohen. Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science*, pages 190–200. IEEE, 1994.

[Con71] J. H̃. Conway. *Regular Algebra and Finite Machines*. Chapman & Hall, 1971.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Third ACM Symp. Theory of Computing*, pages 151–158, 1971.

[Coo11] William J Cook. In *Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2011.

[CP90] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. In *Operations Research Letters*, volume 9, pages 375–382, 1990.

[CP05] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*. Springer, second edition, 2005.

[CP18] Phillip Compeau and P. A. Pevzner. *Bioinformatics Algorithms: An Active Learning Approach*. Active Learning, third edition, 2018.

[CPARS18] Haochen Chen, Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. A tutorial on network embeddings. *arXiv preprint arXiv:1808.02590*, 2018.

[CPW98] B. Chen, C. Potts, and G. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 21–169. Kluwer, 1998.

[CR76] J. Cohen and M. Roth. On the implementation of Strassen's fast multiplication algorithm. *Acta Informatica*, 6:341–355, 1976.

[CR99] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[CR03] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.

[CS93] J. Conway and N. Sloane. *Sphere Packings, Lattices, and Groups*. Springer-Verlag, 1993.

[CSG05] A. Caprara and J. Salazar-González. Laying out sparse graphs with provably minimum bandwidth. *INFORMS J. Computing*, 17:356–373, 2005.

[CT65] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CT92] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80:1412–1434, 1992.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, pages 251–280, 1990.

[CW16] Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. *ACM Transactions on Algorithms (TALG)*, 12(4):45, 2016.

[Dan63] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[Dan94] V. Dancik. Expected length of longest common subsequences. PhD. thesis, Univ. of Warwick, 1994.

[DB74] G. Dahlquist and A. Bjorck. *Numerical Methods*. Prentice-Hall, 1974.

[DB86] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software — Practice and Experience*, 16:575–601, 1986.

[dBDK+98] M. de Berg, O. Devillers, M. Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. *Theoretical Computer Science*, 31:613–628, 1998.

[dBvKOS08] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition, 2008.

[DCSL18] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 246–256. ACM, 2018.

[DDS09] Christopher Dyken, Morten Dahlen, and Thomas Sevaldrud. Simultaneous curve simplification. *Journal of geographical systems*, 11(3):273–289, 2009.

[Dem02] Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[Den05] L. Y. Deng. Efficient and portable multiple recursive generators of large order. *ACM Trans. / on Modeling and Computer Simulation*, 15:1–13, 2005.

[Dey06] T. Dey. *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis*. Cambridge Univ. Press, 2006.

[DF79] E. Denardo and B. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27:161–186, 1979.

[DF08] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, pages 537–546. ACM, 2008.

[DF12] Rodney G. Downey and Michael Ralph Fellows. *Parameterized Complexity*. Springer Science & Business Media, 2012.

[DFJ54] G. Dantzig, D. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[DFU11] Chandan Dubey, Uriel Feige, and Walter Unger. Hardness results for approximating the bandwidth. *Journal of Computer and System Sciences*, 77(1):62–90, 2011.

[DGH+02] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289:69–83, 2002.

[DGKK79] R. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[DH92] D. Du and F. Hwang. A proof of Gilbert and Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7:121–135, 1992.

[DH00] Dingzhu Du and Frank Hwang. *Combinatorial Group Testing and its Applications*, volume 12. World Scientific, 2000.

[DHS00] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, second edition, 2000.

[Die04] M. Dietzfelbinger. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES Is in P"*. Springer, 2004.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DIM16] Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, 2016.

[DIM18] Maxence Delorme, Manuel Iori, and Silvano Martello. Bpplib: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250, 2018.

[DJ92] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[Dji00] H. Djidjev. Computing the girth of a planar graph. In *Proc. 27th Int. Colloquium on Automata, Languages and Programming (ICALP)*, pages 821–831, 2000.

[DJP04] E. Demaine, T. Jones, and M. Patrascu. Interpolation search for nonindependent data. In *Proc. 15th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 522–523, 2004.

[DKR10] Prasun Dutta, S. Pratik Khastgir, and Anushree Roy. Steiner trees and spanning trees in six-pin soap films. *American Journal of Physics*, 78(2):215–221, 2010.

[DL76] D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Computing*, 5:181–186, 1976.

[DLR79] D. Dobkin, R. Lipton, and S. Reiss. Linear programming is log-space hard for P. *Info. Processing Letters*, 8:96–97, 1979.

[DM80] D. Dobkin and J. Ĩ. Munro. Determining the mode. *Theoretical Computer Science*, 12:255–263, 1980.

[DM97] K. Daniels and V. Milenkovic. Multiple translational containment. part I: an approximation algorithm. *Algorithmica*, 19:148–182, 1997.

[DMBS79] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, 1979.

[dMPF09] Francisco de Moura Pinto and Carla Maria Dal Sasso Freitas. Fast medial axis transform for planar domains with general boundaries. In *2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 96–103. IEEE, 2009.

[DMR97] K. Daniels, V. Milenkovic, and D. Roth. Finding the largest area axisparallel rectangle in a polygon. *Computational Geometry: Theory and Applications*, 7:125–148, 1997.

[DN07] P. D'Alberto and A. Nicolau. Adaptive Strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing*, pages 284–292, 2007.

[DN09] Paolo D'Alberto and Alexandru Nicolau. Adaptive winograd's matrix multiplications. *ACM Transactions on Mathematical Software (TOMS)*, 36(1):3, 2009.

[DNMB18] Wouter De Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory Social Network Analysis with Pajek: Revised and Expanded Edition for Updated Software*, volume 46. Cambridge University Press, 2018.

[DP73] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, December 1973.

[DP18] Samuel Dittmer and Igor Pak. Counting linear extensions of restricted posets. *arXiv preprint arXiv:1802.06312*, 2018.

[DPS02] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34:313–356, 2002.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2008.

[DR90] N. Dershowitz and E. Reingold. Calendrical calculations. *Software — Practice and Experience*, 20:899–928, 1990.

[DR02] N. Dershowitz and E. Reingold. *Calendrical Tabulations: 1900–2200*. Cambridge University Press, 2002.

[DRR+95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 247–258, 1995.

[DS05] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, pages 439–485, 2005.

[DSR00] D. Du, J. Smith, and J. Rubinstein. *Advances in Steiner Trees*. Kluwer, 2000.

[DT04] M. Dorigo and T. Stutzle. *Ant Colony Optimization*. MIT Press, 2004.

[dVS82] G. de V. Smit. A comparison of three string matching algorithms. *Software—Practice and Experience*, 12:57–66, 1982.

[dVV03] S. de Vries and R. Vohra. Combinatorial auctions: A survey. *Informs J. Computing*, 15:284–309, 2003.

[DY94] Y. Deng and C. Yang. Waring's problem for pyramidal numbers. *Science in China (Series A)*, 37:377–383, 1994.

[DZ99] D. Dor and U. Zwick. Selecting the median. *SIAM J. Computing*, pages 1722–1758, 1999.

[DZ01] D. Dor and U. Zwick. Median selection requires $(2+\epsilon)n$ comparisons. *SIAM J. Discrete Math.*, 14:312–325, 2001.

[Ebe88] J. Ebert. Computing Eulerian trails. *Info. Proc. Letters*, 28:93–97, 1988.

[ECW92] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[Ede87] H. Edelsbrunner. *Algorithms for Combinatorial Geometry*. Springer-Verlag, 1987.

[Ede06] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, 2006.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1965.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[EE99] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. *Disc. Comp. Geometry*, 22:569–592, 1999.

[EG60] P. Erdős and T. Gallai. Graphs with prescribed degrees of vertices. *Mat. Lapok (Hungarian)*, 11:264–274, 1960.

[EG89] H. Edelsbrunner and L. Guibas. Topologically sweeping an arrangement. *J. Computer and System Sciences*, 38:165–194, 1989.

[EG91] H. Edelsbrunner and L. Guibas. Corrigendum: Topologically sweeping an arrangement. *J. Computer and System Sciences*, 42:249–251, 1991.

[EGI98] David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*, pages 181–205. CRC Press, 1998.

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification: a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.

[EH10] Herbert Edelsbrunner and John Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2010.

[EHM+12] Yuval Emek, Magnús M. Halldórsson, Yishay Mansour, Boaz Patt- Shamir, Jaikumar Radhakrishnan, and Dror Rawitz. Online set packing. *SIAM Journal on Computing*, 41(4):728–746, 2012.

[EJ73] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman. *Math. Programming*, 5:88–124, 1973.

[EK72] J. Edmonds and R. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[EK10] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets*. Cambridge University Press, 2010.

[EKA84] M. I. Edahiro, I. Kokubo, and T. Asano. A new point location algorithm and its practical efficiency – comparison with existing algorithms. *ACM Trans./ Graphics*, 3:86–109, 1984.

[EKS83] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans./ on Information Theory*, IT-29:551–559, 1983.

[EM94] H. Edelsbrunner and E. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43–72, 1994.

[ENSS98] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[Epp98] D. Eppstein. Finding the $k$ shortest paths. *SIAM J. Computing*, 28:652–673, 1998.

[ES86] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete & Computational Geometry*, 1:25–44, 1986.

[ESS93] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Computing*, 22:418–429, 1993.

[ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proc. IEEE Visualization '96*, pages 319–326, 1996.

[Eve79] G. Everstine. A comparison of three resequencing algorithms for the reduction of matrix profile and wave-front. *Int. J. Numerical Methods in Engr.*, 14:837–863, 1979.

[Eve11] Shimon Even. *Graph Algorithms*. Cambridge University Press, second edition, 2011.

[Fá48] I. Fáry. On straight line representation of planar graphs. *Acta. Sci. Math. Szeged*, 11:229–233, 1948.

[FAKM14] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[Fei98] U. Feige. A threshold of ln $n$ for approximating set cover. *J. ACM*, 45:634–652, 1998.

[FF62] L. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FG95] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *Proc. 3rd Israel Symp. on Theory of Computing and Systems*, pages 182–189, 1995.

[FH06] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums for convex polyhedra with applications. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Springer, 2001.

[FHW07] E. Fogel, D. Halperin, and C. Weibel. On the exact maximum complexity of minkowski sums of convex polyhedra. In *Proc. 23rd Symp. Computational Geometry*, pages 319–326, 2007.

[FHW12] Efi Fogel, Dan Halperin, and Ron Wein. *CGAL Arrangements and Their Applications: A Step-by-Step Guide*, volume 7. Springer Science & Business Media, 2012.

[FJ05] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2005.

[FJMO93] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proc. 4th 7th Symp. Discrete Algorithms (SODA)*, pages 145–154, 1993.

[FK10] Alireza Fathi and John Krumm. Detecting road intersections from gps traces. In *International Conference on Geographic Information Science*, pages 56–69. Springer, 2010.

[FK15] Alan Frieze and Michał Karoński. *Introduction to Random Graphs*. Cambridge University Press, 2015.

[FKN+08] Michael R. Fellows, Christian Knauer, Naomi Nishimura, Prabhakar Ragde, F. Rosamond, Ulrike Stege, Dimitrios M. Thilikos, and Sue Whitesides. Faster fixed-parameter tractable algorithms for matching and packing problems. *Algorithmica*, 52(2):167–176, 2008.

[FL07] Uriel Feige and James R. Lee. An improved approximation ratio for the minimum linear arrangement problem. *Information Processing Letters*, 101(1):26–29, 2007.

[Fle74] H. Fleischner. The square of every two-connected graph is Hamiltonian. *J. Combinatorial Theory, B*, 16:29–34, 1974.

[Flo62] R. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 7:345, 1962.

[Flo64] R. Floyd. Algorithm 245 (treesort). *Communications of the ACM*, 18:701, 1964.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cacheoblivious algorithms. In *Proc. 40th Symp. Foundations of Computer Science*, 1999.

[FM71] M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE 12th Symp. on Switching and Automata Theory*, pages 129–131, 1971.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, 1982.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9, 2004.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For13] Lance Fortnow. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press, 2013.

[For18] S. Fortune. Voronoi diagrams and Delauney triangulations. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 705–722. CRC Press, 2018.

[FPR99] P. Festa, P. Pardalos, and M. Resende. Feedback set problems. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume A. Kluwer, 1999.

[FPR01] P. Festa, P. Pardalos, and M. Resende. Algorithm 815: Fortran subroutines for computing approximate solution to feedback set problems using GRASP. *ACM Transactions on Mathematical Software*, 27:456–464, 2001.

[FR75] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[FR94] M. Fürer and B. Raghavachari. Approximating the minimum-degree Steiner tree to within one of optimal. *J. Algorithms*, 17:409–423, 1994.

[Fre62] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[Fre76] M. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley, 2003.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[FT87] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[FvW93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th ACM Symp. Computational Geometry*, pages 163–172, 1993.

[FW77] S. Fiorini and R. Wilson. *Edge-Colourings of Graphs*. Research Notes in Mathematics 16, Pitman, 1977.

[FW93] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *J. Computer and System Sci.*, 47:424–436, 1993.

[FWH04] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th European Symposium on Algorithms (ESA '04)*, pages 664–676, 2004.

[Gab77] H. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Computing*, 6:139–150, 1977.

[GAD+13] Jared L. Gearhart, Kristin L. Adair, Richard J. Detry, Justin D. Durfee, Katherine A. Jones, and Nathaniel Martin. Comparison of open-source linear programming solvers. *Sandia National Laboratories, SAND2013–8847*, 2013.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *ACM Computing Surveys*, 18:23–38, 1986.

[Gal90] K. Gallivan. *Parallel Algorithms for Matrix Computations*. SIAM, 1990.

[GALL13] Mukulika Ghosh, Nancy M. Amato, Yanyan Lu, and Jyh-Ming Lien. Fast approximate convex decomposition using relative concavity. *Computer-Aided Design*, 45(2):494–504, 2013.

[Gas03] S. Gass. *Linear Programming: Methods and Applications*. Dover, fifth edition, 2003.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, second edition, 1991.

[GE19] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *arXiv preprint arXiv:1905.09749*, 2019.

[Gen06] James E Gentle. *Random Number Generation and Monte Carlo Methods*. Springer Science & Business Media, 2006.

[GGJ77] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.*, 32:835–859, 1977.

[GGJK78] M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity results for bandwidth minimization. *SIAM J. Appl. Math.*, 34:477–495, 1978.

[GH85] R. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1985.

[GH06] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers and Operations Research*, 33:2547–2562, 2006.

[Gha16] Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 270–277, 2016.

[GHHP13] Philippe Galinier, Jean-Philippe Hamiez, Jin-Kao Hao, and Daniel Porumbel. Recent advances in graph vertex coloring. In *Handbook of optimization*, pages 505–528. Springer, 2013.

[GHMS93] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[GHR95] R. Greenlaw, J. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, 1995.

[GI89] D. Gusfield and R. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.

[GI91] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, 1991.

[Gin18] M. Ginsberg. *Factor Man*. Zowie Press, 2018.

[GJ77] M. Garey and D. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM J. Appl. Math.*, 32:826–834, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[GJM02] M. Goldwasser, D. Johnson, and C. McGeoch, editors. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59. AMS, 2002.

[GJPT78] M. Garey, D. Johnson, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Info. Proc. Letters*, 7:175–180, 1978.

[GK95] A. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Programming*, 71:153–177, 1995.

[GK98] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20:374–387, 1998.

[GKK74] F. Glover, D. Karney, and D. Klingman. Implementation and computational comparisons of primal-dual computer codes for minimum-cost network flow problems. *Networks*, 4:191–212, 1974.

[GKP89] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[GKS95] S. Gupta, J. Kececioglu, and A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Computational Biology*, 2:459–472, 1995.

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[GKT05] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. 31st Int. Conf on Very Large Data Bases*, pages 721–732, 2005.

[GKW06] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proc. 8th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2006.

[GL96] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[GM86] G. Gonnet and J. Ĩ. Munro. Heaps on heaps. *SIAM J. Computing*, 15:964–971, 1986.

[GM91] S. Ghosh and D. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computing*, 20:888–910, 1991.

[GM12] Gary Gordon and Jennifer McNulty. *Matroids: a geometric introduction*. Cambridge University Press, 2012.

[GMPV06] F. Gomes, C. Meneses, P. Pardalos, and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers and Operations Research*, 33:3520–3534, 2006.

[GO95] Anka Gajentaan and Mark H Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5(3):165–185, 1995.

[GO14] Gene H. Golub and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Elsevier, 2014.

[Goe97] M. Goemans. Semidefinite programming in combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[Gol93] L. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures*. Cambridge University Press, 1993.

[Gol97] A. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997.

[Gol01] A. Goldberg. Shortest path algorithms: Engineering aspects. In *12th International Symposium on Algorithms and Computation*, number 2223 in LNCS, pages 502–513. Springer, 2001.

[Gol04] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. North Holland, second edition, 2004.

[Gon08] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Gon18] T. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman-Hall/CRC Press, second edition, 2018.

[GP68] E. Gilbert and H. Pollak. Steiner minimal trees. *SIAM J. Applied Math.*, 16:1–29, 1968.

[GP79] B. Gates and C. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[GP07] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

[GPS76] N. Gibbs, W. Poole, and P. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans./ Math. Software*, 2:322–330, 1976.

[Gra53] F. Gray. Pulse code communication. US Patent 2632058, March 17, 1953.

[Gra72] R. Graham. An efficient algorithm for determining the convex hull of a finite planar point set. *Info. Proc. Letters*, 1:132–133, 1972.

[Gri89] D. Gries. *The Science of Programming*. Springer-Verlag, 1989.

[GS62] D. Gale and L. Shapely. College admissions and the stability of marriages. *American Math. Monthly*, 69:9–14, 1962.

[GT94] T. Gensen and B. Toft. *Graph Coloring Problems*. John Wiley & Sons, 1994.

[GT01] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory*. Courier Corporation, 2001.

[GT14] Andrew V. Goldberg and Robert E. Tarjan. Efficient maximum flow algorithms. *Communications of the ACM*, 57(8):82–89, 2014.

[GTG14] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Java*. John Wiley & Sons, sixth edition, 2014.

[Gup66] R. P. Gupta. The chromatic index and the degree of a graph. *Notices of the Amer. Math. Soc.*, 13:719, 1966.

[Gus94] D. Gusfield. Faster implementation of a shortest superstring approximation. *Info. Processing Letters*, 51:271–274, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GW95] M. Goemans and D. Williamson. 878-approximation algorithms for MAX CUT and MAX 2SAT. *J. ACM*, 42:1115–1145, 1995.

[GW96] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pages 66–70, 1996.

[GW97] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European J. Operational Research*, 101, 1997.

[Ham87] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover, second edition, 1987.

[Has82] H. Hastad. Clique is hard to approximate within $n^{1}$ '. *Acta Mathematica*, 182:105–142, 182.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12:381–402, 1980.

[HDD03] M. Hilgemeier, N. Drechsler, and R. Drechsler. Fast heuristics for the edge coloring of large graphs. In *Proc. Euromicro Symp. on Digital Systems Design*, pages 230–239, 2003.

[HdlT01] J. Holm, K. de lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[Hel01] M. Held. VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Computational Geometry: Theory and Applications*, 18:95–123, 2001.

[HFN05] H. Hyyro, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HG97] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes, 1997.

[HH00] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th International Workshop on Algorithm Engineering (WAE), LNCS v. 1982*, pages 171–182, 2000.

[HH09] Idit Haran and Dan Halperin. An experimental study of point location in planar arrangements in CGAL. *Journal of Experimental Algorithmics (JEA)*, 13:3, 2009.

[HHL09] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review lLetters*, 103(15):150502, 2009.

[HHS98] T. Haynes, S. Hedetniemi, and P. Slater. *Fundamentals of Domination in Graphs*. CRC Press, Boca Raton, 1998.

[HIKP12] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. Simple and practical algorithm for sparse fourier transform. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1183–1194, 2012.

[Hir75] D. Hirschberg. A linear-space algorithm for computing maximum common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[HK73] J. Hopcroft and R. Karp. An $n^{5\,3}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HK90] D. P. Huttenlocher and K. Kedem. Computing the minimum Hausdorff distance for point sets under translation. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 340–349, 1990.

[HK11] Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata — a survey. *Information and Computation*, 209(3):456–470, 2011.

[HKH12] Michael Hemmer, Michal Kleinbort, and Dan Halperin. Improved implementation of point location in general two-dimensional subdivisions. In *European Symposium on Algorithms*, pages 611–623. Springer, 2012.

[HLD04] W. Hörmann, J. Leydold, and G. Derfinger. *Automatic Nonuniform Random Variate Generation*. Springer, 2004.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, pages 207–218. Lecture Notes in Computer Science, Vol. 158, 1983.

[HM99] X. Huang and A. Madan. Cap3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[HMS03] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. In *Proc. 5th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2003.

[HMU06] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2006.

[HNP91] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, 1991.

[HNSS18] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *Journal of Experimental Algorithmics (JEA)*, 23(1):1–8, 2018.

[Hoa61] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4:321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[Hoc96] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing, 1996.

[Hof82] C. M. Hoffmann. *Group-theoretic algorithms and graph isomorphism*. Lecture Notes in Computer Science. Springer-Verlag Inc., 1982.

[Hol75] J. H̃. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Hol81] I. Holyer. The NP-completeness of edge colorings. *SIAM J. Computing*, 10:718–720, 1981.

[Hol92] J. H̃. Holland. Genetic algorithms. *Scientific American*, 267(1):66–72, 1992.

[Hop71] J. Hopcroft. An $n$ log $n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software — Practice and Experience*, 10:501–506, 1980.

[HP73] F. Harary and E. Palmer. *Graphical Enumeration*. Academic Press, New York, 1973.

[HP09] Karla Hoffman and Manfred Padberg. Set covering, packing and partitioning problems. *Encyclopedia of Optimization*, pages 3482–3486, 2009.

[HPS+05] M. Holzer, G. Prasinos, F. Schulz, D. Wagner, and C. Zaroliagis. Engineering planar separator algorithms. In *Proc. 13th European Symp. on Algorithms (ESA)*, pages 628–637, 2005.

[HRW92] R. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North Holland, 1992.

[HRW17] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938, 2017.

[HS77] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[HS94] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker algorithm for line simplification. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 383–384, 1994.

[HS98] J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulae in O($n$ log$^*$ $n$) time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998.

[HS99] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Computing*, 28:2215–2256, 1999.

[HS18] D. Halperin and M. Sharir. Arrangements. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 723–762. CRC Press, 2018.

[HSS87] J. Hopcroft, J. Schwartz, and M. Sharir. *Planning, Geometry, and Complexity of Robot Motion*. Ablex Publishing, Norwood NJ, 1987.

[HSS07] R. Hardin, N. Sloane, and W. Smith. Maximum volume spherical codes. http://www.research.att.com/~njas/maxvolumes/, 2007.

[HSS18] D. Halperin, O. Salzman, and M. Sharir. Algorithmic motion planning. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1311–1343. CRC Press, 2018.

[HSWW05] M. Holzer, F. Schultz, D. Wagner, and T. Willhalm. Combining speedup techniques for shortest-path computations. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HT73a] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2:135–158, 1973.

[HT73b] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 1973.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[Hub06] M. Huber. Fast perfect sampling from linear extensions. *Disc. Math.*, 306:420–428, 2006.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40:1098–1101, 1952.

[HUW02] E. Haunschmid, C. Ueberhuber, and P. Wurzinger. Cache oblivious high performance algorithms for matrix multiplication. Vienna University of Technology, Tech. Report AURORA TR2002-08, 2002.

[HVDH19] David Harvey and Joris Van Der Hoeven. Integer multiplication in time $O(n \log n)$, 2019.

[HVDHL16] David Harvey, Joris Van Der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1–30, 2016.

[HW74] J. Ẽ. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[HWA+03] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. PCAP: A wholegenome assembly program. *Genome Research*, 13:2164–2170, 2003.

[IK75] O. Ibarra and C. Kim. Fast approximation algorithms for knapsack and sum of subset problems. *J. ACM*, 22:463–468, 1975.

[IMS18] P. Indyk, J. Matousek, and A. Sidiropoulous. Low-distortion embeddings of finite metric spaces. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 211–232. CRC Press, 2018.

[Ind98] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proc. 39th Symp. Foundations of Computer Science*, 1998.

[IR78] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7:413–423, 1978.

[IR09] Costas S. Iliopoulos and M. Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory of Computing Systems*, 45(2):355–371, 2009.

[Ita78] A. Itai. Two commodity flow. *J. ACM*, 25:596–611, 1978.

[Iwa16] Yoichi Iwata. Linear-time kernelization for feedback vertex set. *arXiv preprint arXiv:1608.01463*, 2016.

[JÓ92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. Symp. Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JAMS91] D. Johnson, C. Aragon, C. McGeoch, and D. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. In *Operations Research*, volume 39, pages 378–406, 1991.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Letters*, 2:18–21, 1973.

[JB19] Riko Jacob and Gerth Stolting Brodal. Dynamic planar convex hull. *arXiv preprint arXiv:1902.11169*, 2019.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[JLR00] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs*. John Wiley & Sons, 2000.

[JM93] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12. American Mathematics Society, 1993.

[JM12] Michael Jünger and Petra Mutzel. *Graph drawing software*. Springer Science & Business Media, 2012.

[Joh63] S. M. Johnson. Generation of permutations by adjacent transpositions. *Math. Computation*, 17:282–285, 1963.

[Joh74] D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 9:256–278, 1974.

[Jon86] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[Jos12] N. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, second edition, 2012.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Computing*, 22:1117–1141, 1993.

[JS91] Brian Johnson and Ben Shneiderman. *Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures*. IEEE, 1991.

[JS01] A. Jagota and L. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *J. Heuristics*, 7:1381–1231, 2001.

[JSV04] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM (JACM)*, 51(4):671–697, 2004.

[JT96] D. Johnson and M. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26. AMS, 1996.

[JWWZ18] Daniel Juhl, David M. Warme, Pawel Winter, and Martin Zachariasen. The geosteiner software package for computing steiner trees in the plane: an updated computational study. *Mathematical Programming Computation*, 10(4):487–532, 2018.

[KA03] P. Ko and S. Aluru. Space-efficient linear time construction of suffix arrays,. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, pages 200–210. Springer-Verlag LNCS, 2003.

[KA10] S.R. Kodituwakku and U. S. Amarasinghe. Comparison of lossless data compression algorithms for text data. *Indian Journal of Computer Science and Engineering*, 1(4):416–425, 2010.

[Kah67] D. Kahn. *The Code Breakers: The Story of Secret Writing*. Macmillan, New York, 1967.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kar96] H. Karloff. How good is the Goemans-Williamson MAX CUT algorithm? In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 427–434, 1996.

[Kar00] D. Karger. Minimum cuts in near-linear time. *J. ACM*, 47:46–76, 200.

[KBFS12] Alexander Kröller, Tobias Baumgartner, Sándor P. Fekete, and Christiane Schmidt. Exact solutions and bounds for general art gallery problems. *Journal of Experimental Algorithmics (JEA)*, 17:2–3, 2012.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[Kei00] M. Keil. Polygon decomposition. In J. R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. Elsevier, 2000.

[KF11] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[Kir79] D. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th IEEE Symp. Foundations of Computing*, pages 28–35, 1979.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Computing*, 12:28–35, 1983.

[KKT95] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42:321–328, 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, 1970.

[KLM+14] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm. In *Inequalities III*, pages 159–172. Academic Press, 1972.

[KM95] J. D̃. Kececioglu and E. W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, 1995.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[KMP+04] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symp. on Algorithms (ESA '04)*, pages 702–713. www.mpiinf. mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps, 2004.

[KMR97] David Karger, Rajeev Motwani, and Gurumurthy D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

[KMS97] S. Khanna, M. Muthukrishnan, and S. Skiena. Efficiently partitioning arrays. In *Proc. ICALP '97*, volume 1256, pages 616–626. Springer-Verlag LNCS, 1997.

[KMS98] János Komlós, Yuan Ma, and Endre Szemerédi. Matching nuts and bolts in $O(n \log n)$ time. *SIAM Journal on Discrete Mathematics*, 11(3):347–372, 1998.

[Knu94] D. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, 1994.

[Knu97a] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, third edition, 1997.

[Knu97b] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1997.

[Knu98] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[Knu11] D. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011.

[Knu15] D. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multi-digit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1963.

[Koe05] H. Koehler. A contraction algorithm for finding minimal feedback sets. In *Proc. 28th Australasian Computer Science Conference (ACSC)*, pages 165–174, 2005.

[KOS91] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 74–77, 1991.

[KP98] J. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proc. 2nd Workshop on Algorithm Engineering*, pages 121–132, 1998.

[KPP04] H. Kellerer, U. Pferschy, and P. Pisinger. *Knapsack Problems*. Springer, 2004.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31:249–260, 1987.

[KR91] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *J. Comp. Sys. Sci.*, 42:288–306, 1991.

[KR08] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within $2$-$\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

[Kru56] J. B̃. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 7:48–50, 1956.

[KS74] D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2:153–157, 1974.

[KS85] M. Keil and J. R̃. Sack. Minimum decomposition of geometric objects. *Machine Intelligence and Pattern Recognition*, 2:197–216, 1985.

[KS86] D. Kirkpatrick and R. Siedel. The ultimate planar convex hull algorithm? *SIAM J. Computing*, 15:287–299, 1986.

[KS99] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[KS02] M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. *Int. J. Comput. Geometry Appl.*, 12:181–192, 2002.

[KS05a] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Info. Proc. Letters*, 93:13–17, 2005.

[KS05b] J. Kelner and D. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Electronic Colloquim on Computational Complexity*, 156:17, 2005.

[KS07] H. Kautz and B. Selman. The state of SAT. *Disc. Applied Math.*, 155:1514–1524, 2007.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[KSBD07] H. Kautz, B. Selman, R. Brachman, and T. Dietterich. *Satisfiability Testing*. Morgan and Claypool, 2007.

[KSPP03] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Symp. Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

[KST93] J. Köbler, U. Schöning, and J. Túran. *The Graph Isomorphism Problem: Its Structural Complexity*. Birhauser, 1993.

[KSV97] D. Keyes, A. Sameh, and V. Venkatarishnan. *Parallel Numerical Algorithms*. Springer, 1997.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2006.

[Kur30] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[KW01] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Springer-Verlag, 2001.

[Kwa62] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[LA04] J. Leung and J. Anderson, editors. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press/Chapman & Hall, 2004.

[LA06] J. Lien and N. Amato. Approximate convex decomposition of polygons. *Computational Geometry: Theory and Applications*, 35:100–123, 2006.

[Lam92] J.-L. Lambert. Sorting the sums $(x_i + y_j)$ in $O(n^2)$ comparisons. *Theoretical Computer Science*, 103:137–141, 1992.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer, 1991.

[Lau98] J. Laumond. *Robot Motion Planning and Control*. Springer-Verlag, Lectures Notes in Control and Information Sciences, Volume 229, 1998.

[LaV06] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[Law11] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Publications, 2011.

[LD03] R. Laycock and A. Day. Automatically generating roof models from building footprints. In *Proc. 11th Int. Conf. Computer Graphics, Visualization and Computer Vision (WSCG)*, 2003.

[L'E12] Pierre L'Ecuyer. Random number generation. In *Handbook of Computational Statistics*, pages 35–71. Springer, 2012.

[Lec95] T. Lecroq. Experimental results on string matching algorithms. *Software — Practice and Experience*, 25:727–765, 1995.

[Lee82] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans./ Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[Len87a] T. Lengauer. Efficient algorithms for finding minimum spanning forests of hierarchically defined graphs. *J. Algorithms*, 8, 1987.

[Len87b] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[Len89] T. Lengauer. Hierarchical planarity testing algorithms. *J. ACM*, 36(3):474–509, July 1989.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.

[LL96] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM J. Experimental Algorithmics*, 1, 1996.

[LL99] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31:66–104, 1999.

[LLCC12] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Transactions on Computers*, 62(10):1906–1916, 2012.

[LLK83] J. K̃. Lenstra, E. L. Lawler, and A. Rinnooy Kan. *Theory of Sequencing and Scheduling*. John Wiley & Sons, 1983.

[LLKS85] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[LLS92] L. Lam, S.-W. Lee, and C. Suen. Thinning methodologies–a comprehensive survey. *IEEE Trans./ Pattern Analysis and Machine Intelligence*, 14:869–885, 1992.

[LMK18] M. Lin, D. Manocha, and Y. Kim. Collision and proximity queries. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1029–1056. CRC Press, 2018.

[LMM+18] Daniel Lokshtanov, Pranabendu Misra, Joydeep Mukherjee, Geevarghese Philip, Fahad Panolan, and Saket Saurabh. A 2-approximation algorithm for feedback vertex set in tournaments. *arXiv preprint arXiv:1809.08437*, 2018.

[LMS06] L. Lloyd, A. Mehler, and S. Skiena. Identifying co-referential names across large corpora. In *Combinatorial Pattern Matching (CPM 2006)*, pages 12–23. Lecture Notes in Computer Science, v.4009, 2006.

[LP02] W. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.

[LP07] A. Lodi and A. Punnen. TSP software. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, pages 737–749. Springer, 2007.

[LP09] László Lovász and Michael D. Plummer. *Matching Theory*, volume 367. American Mathematical Soc., 2009.

[LP16] Adi Livnat and Christos H. Papadimitriou. Sex as an algorithm: the theory of evolution under the lens of computation. *Commun. ACM*, 59(11):84–93, 2016.

[LPW79] T. Lozano-Perez and M. Wesley. An algorithm for planning collision-free paths among polygonal obstacles. *Comm. ACM*, 22:560–570, 1979.

[LR93] K. Lang and S. Rao. Finding near-optimal cuts: An empirical evaluation. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 212–221, 1993.

[LRSV18] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[LS87] V. Lumelski and A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 3:403–430, 1987.

[LS95] Y.-L. Lin and S. Skiena. Algorithms for square roots of graphs. *SIAM J. Discrete Mathematics*, 8:99–118, 1995.

[LSCK02] P. L'Ecuyer, R. Simard, E. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002.

[LST14] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A backto-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. SIAM, 2014.

[LSZ+15] Liang Liu, Yuning Song, Haiyang Zhang, Huadong Ma, and Athanasios V Vasilakos. Physarum optimization: A biology-inspired algorithm for the steiner tree problem in networks. *IEEE Transactions on Computers*, 64(3):818–831, 2015.

[LT79] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:346–358, 1979.

[LT80] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM J. Computing*, 9:615–626, 1980.

[Luc91] E. Lucas. *Récréations Mathématiques*. Gauthier-Villares, Paris, 1891.

[Luk80] E. M. Luks. Isomorphism of bounded valence can be tested in polynomial time. In *Proc. of the 21st Annual Symposium on Foundations of Computing*, pages 42–49. IEEE, 1980.

[LV88] G. Landau and U. Vishkin. Fast string matching with $k$ differences. *J. Comput. System Sci.*, 37:63–78, 1988.

[LV97] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New York, second edition, 1997.

[LW77] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[LW88] T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM J. Computing*, 17:1063–1080, 1988.

[M+13] Aranyak Mehta et al. Online matching and ad allocation. *Foundations and Trends in Theoretical Computer Science*, 8(4):265–368, 2013.

[Mah76] S. Maheshwari. Traversal marker placement problems are NP-complete. Technical Report CU-CS-09276, Department of Computer Science, University of Colorado, Boulder, 1976.

[Mai78] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[Mak02] R. Mak. *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing*. Prentice Hall, 2002.

[Man89] U. Manber. *Introduction to Algorithms*. Addison-Wesley, 1989.

[Man12] Toufik Mansour. *Combinatorics of Set Partitions*. Chapman & Hall/CRC Press, 2012.

[MAR+17] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*, 2017.

[Mat87] D. W. Matula. Determining edge connectivity in $O(nm)$. In *28th Ann. Symp. Foundations of Computer Science*, pages 249–251. IEEE, 1987.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[McK81] B.McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[MDS01] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, second edition, 2001.

[Men27] K. Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[Mey01] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 2001.

[MF00] Z. Michalewicz and D. Fogel. *How To Solve It: Modern Heuristics*. Springer, 2000.

[MG92] J. Misra and D. Gries. A constructive proof of Vizing's theorem. *Info. Processing Letters*, 41:131–133, 1992.

[MG07] Jiri Matousek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer Science & Business Media, 2007.

[MH78] R. Merkle and M. Hellman. Hiding and signatures in trapdoor knapsacks. *IEEE Trans./ Information Theory*, 24:525–530, 1978.

[MH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[Mil76] G. Miller. Riemann's hypothesis and tests for primality. *J. Computer and System Sciences*, 13:300–317, 1976.

[Mil97] V. Milenkovic. Multiple translational containment. part II: exact algorithms. *Algorithmica*, 19:183–218, 1997.

[Min78] H. Minc. *Permanents*, volume 6 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, 1978.

[Mit99] J. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-mst, and related problems. *SIAM J. Computing*, 28:1298–1309, 1999.

[MKT07] E. Mardis, S. Kim, and H. Tang, editors. *Advances in Genome Sequencing Technology and Algorithms*. Artech House Publishers, 2007.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 36(11):2227–2240, 2014.

[MLL16] Javier Minguez, Florant Lamiraux, and Jean-Paul Laumond. Motion planning and obstacle avoidance. In *Springer Handbook of Robotics*, pages 1177–1202. Springer, 2016.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. *SIAM J. Computing*, pages 935–948, 1993.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[MMI72] D. Matula, G. Marble, and J. Isaacson. Graph coloring algorithms. In R. C. Read, editor, *Graph Theory and Computing*, pages 109–122. Academic Press, 1972.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans./ on Modeling and Computer Simulation*, 8:3–30, 1998.

[MN99] K. Mehlhorn and S. Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MN07] V. Makinen and G. Navarro. Compressed full text indexes. *ACM Computing Surveys*, 39, 2007.

[MNM+16] Thomas Monz, Daniel Nigg, Esteban A. Martinez, Matthias F. Brandl, Philipp Schindler, Richard Rines, Shannon X. Wang, Isaac L. Chuang, and Rainer Blatt. Realization of a scalable Shor algorithm. *Science*, 351(6277):1068–1070, 2016.

[MO63] L. E. Moses and R. V. Oakford. *Tables of Random Permutations*. Stanford University Press, 1963.

[Moo59] E. F. Moore. The shortest path in a maze. In *Proc. International Symp. Switching Theory*, pages 285–292. Harvard University Press, 1959.

[MOS11] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. A general approach to the analysis of controlled perturbation algorithms. *Computational Geometry*, 44(9):507–528, 2011.

[Mou18] D. Mount. Geometric intersection. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1113–1134. CRC Press, 2018.

[MOV96] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Computer and System Sciences*, 20:18–31, 1980.

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[MPC+06] S. Mueller, D. Papamichial, J. R. Coleman, S. Skiena, and E. Wimmer. Reduction of the rate of poliovirus protein synthesis through large scale codon deoptimization causes virus attenuation of viral virulence by lowering specific infectivity. *J. of Virology*, 80:9687–96, 2006.

[MPT99] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0–1 knapsack problem. *Management Science*, 45:414–424, 1999.

[MPT00] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MR01] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Info. Processing Letters*, 79:281–284, 2001.

[MR06] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In *Proc. 22nd ACM Symp. on Computational Geometry*, pages 1–10, 2006.

[MR10] Nabil H. Mustafa and Saurabh Ray. Improved results on geometric hitting set problems. *Discrete & Computational Geometry*, 44(4):883–895, 2010.

[MRRT53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller.Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, June 1953.

[MS91] B. Moret and H. Shapiro. *Algorithm from P to NP: Design and Efficiency*. Benjamin/Cummings, 1991.

[MS95a] D. Margaritis and S. Skiena. Reconstructing strings from substrings in rounds. In *Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS)*, 1995.

[MS95b] J. S. B. Mitchell and S. Suri. Separation and approximation of polyhedralobjects. *Comput. Geom. Theory Appl.*, 5:95–114, 1995.

[MS00] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Trans./ Mathematical Software*, 26:436–461, 2000.

[MS18] D. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC Press, second edition, 2018.

[MT85] S. Martello and P. Toth. A program for the *0-1* multiple knapsack problem. *ACM Trans. Math. Softw.*, 11(2):135–140, June 1985.

[MT87] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, editor, *Surveys in Combinatorial Optimization*, volume 31 of *Annals of Discrete Mathematics*, pages 213–258. North-Holland, 1987.

[MT90a] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley & Sons, 1990.

[MT90b] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 301–341. MIT Press, 1990.

[MT10] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[MT12] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, 20, 2012.

[MU17] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, second edition, 2017.

[Muc13] Marcin Mucha. Lyndon words and short superstrings. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 958–972. SIAM, 2013.

[Mul94] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, New York, 1994.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[MV80] S. Micali and V. Vazirani. An $O\left(\sqrt{|V|}|E|\right)$ algorithm for finding maximum matchings in general graphs. In *Proc. 21st. Symp. Foundations of Computing*, pages 17–27, 1980.

[MV99] B. McCullough and H. Vinod. The numerical reliability of econometric software. *J. Economic Literature*, 37:633–665, 1999.

[MY07] K. Mehlhorn and C. Yap. *Robust Geometric Computation*. manuscript, http://cs.nyu.edu/yap/book/egc/, 2007.

[Mye86] E. Myers. An O(*nd*) difference algorithm and its variations. *Algorithmica*, 1:514–534, 1986.

[Mye99a] E. Myers. Whole-genome DNA sequencing. *IEEE Computational Engineering and Science*, 3:33–43, 1999.

[Mye99b] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *J. ACM*, 46:395–415, 1999.

[Nav01a] Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[Nav01b] G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.

[NC02] Michael A Nielsen and Isaac Chuang. *Quantum Computation and Quantum Information*. AAPT, 2002.

[Nes13] Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87. Springer Science & Business Media, 2013.

[Neu63] J. V̈on Neumann. Various techniques used in connection with random digits. In A. H. Traub, editor, *John von Neumann, Collected Works*, volume 5. Macmillan, 1963.

[New18] Mark Newman. *Networks*. Oxford University Press, 2018.

[NH19] M. Needham and A. Hodler. *Graph Algorithms: Practical Examples in Apache Spark and Neo4J*. O'Reilly, 2019.

[NI08] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008.

[NLKB11] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 331–342. ACM, 2011.

[Not02] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3:131–144, 2002.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. of Experimental Algorithmics*, 5, 2000.

[NR04] T. Nishizeki and S. Rahman. *Planar Graph Drawing*. World Scientific, 2004.

[NR07] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2007.

[NS07] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge Univ. Press, 2007.

[Nuu95] E. Nuutila. Efficient transitive closure computation in large digraphs. http://www.cs.hut.fi/~enu/thesis.html, 1995.

[NW78] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, second edition, 1978.

[NZ02] S. Näher and O. Zlotowski. Design and implementation of efficient data types for static graphs. In *European Symposium on Algorithms (ESA)*, pages 748–759, 2002.

[NZM91] I. Niven, H. Zuckerman, and H. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, New York, fifth edition, 1991.

[OBSC00] A. Okabe, B. Boots, K. Sugihara, and S. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, 2000.

[Ogn93] R. Ogniewicz. *Discrete Voronoi Skeletons*. Hartung-Gorre Verlag, 1993.

[Omo89] Stephen M. Omohundro. *Five Balltree Construction Algorithms*. International Computer Science Institute Berkeley, 1989.

[O'R85] J. O'Rourke. Finding minimal enclosing boxes. *Int. J. Computer and Information Sciences*, 14:183–199, 1985.

[O'R87] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[O'R01] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 2001.

[Orl13] James B. Orlin. Max flows in $O(nm)$ time, or better. In *Proceedings of the Forty-Fifth annual ACM Symposium on Theory of Computing*, pages 765–774. ACM, 2013.

[OST18] J. O'Rourke, S. Suri, and C. Toth. Polygons. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 787–810. CRC Press, 2018.

[O08] Owen O'Malley. Terabyte sort on apache hadoop. http:// sortbenchmark.org/Yahoo-Hadoop.pdf, 2008.

[Pal14] Katarzyna Paluch. Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring. arXiv 1401.3670, 2014.

[Pan06] R. Panigrahy. *Hashing, Searching, Sketching*. PhD thesis, Stanford University, 2006.

[Pap76a] C. Papadimitriou. The complexity of edge traversing. *J. ACM*, 23:544–554, 1976.

[Pap76b] C. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[Par90] G. Parker. A better phonetic search. *C Gazette*, 5–4, June/July 1990.

[PARS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proc. 20th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, pages 701–710. ACM, 2014.

[Pas97] V. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *Computing Surveys*, s171–209:171–209, 1997.

[Pas03] V. Paschos. Polynomial approximation and graph-coloring. *Computing*, 70:41–86, 2003.

[Pat13] Maurizio Patrignani. Planarity testing and embedding, 2013.

[Pav82] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville MD, 1982.

[PČY+16] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016.

[Pec04] M. Peczarski. New results in minimum-comparison sorting. *Algorithmica*, 40:133–145, 2004.

[Pec07] M. Peczarski. The Ford-Johnson algorithm still unbeaten for less than 47 elements. *Info. Processing Letters*, 101:126–128, 2007.

[Pet03] J. Petit. Experiments on the minimum linear arrangement problem. *ACM J. of Experimental Algorithmics*, 8, 2003.

[PFTV07] W. Press, B. Flannery, S. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[PH80] M. Padberg and S. Hong. On the symmetric traveling salesman problem: a computational study. *Math. Programming Studies*, 12:78–107, 1980.

[PIA78] Y. Perl, A. Itai, and H. Avni. Interpolation search–a log log *n* search. *Comm. ACM*, 21:550–554, 1978.

[Pin16] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, fifth edition, 2016.

[PL94] P. A. Pevzner and R. J. Lipshutz. Towards DNA sequencing chips. In *19th Int. Conf. Mathematical Foundations of Computer Science*, volume 841, pages 143–158, Lecture Notes in Computer Science, 1994.

[PLM06] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans./ Mathematical Software*, 32:1–16, 2006.

[PM88] S. Park and K. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.

[PN18] Shortest Paths and Networks. J. Mitchell. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 811–848. CRC Press, 2018.

[Pol57] G. Polya. *How to Solve It*. Princeton University Press, second edition, 1957.

[Pom84] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarrson, editors, *Advances in Cryptology*, volume 209, pages 169–182. Lecture Notes in Computer Science, Springer-Verlag, 1984.

[PP06] M. Penner and V. Prasanna. Cache-friendly implementations of transitive closure. *ACM J. of Experimental Algorithmics*, 11, 2006.

[PR86] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM J. Computing*, 23:1994, 373–386.

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49:16–34, 2002.

[Pra75] V. Pratt. Every prime has a succinct certificate. *SIAM J. Computing*, 4:214–220, 1975.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[Prü18] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.

[PS85] F. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[PS98] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[PS02] H. Prömel and A. Steger. *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity*. Friedrick Vieweg and Son, 2002.

[PS03] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

[PSL90] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Analysis*, 11:430–452, 1990.

[PSS07] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. In *Proc. 6th Workshop on Experimental Algorithms (WEA), LNCS 4525*, pages 108–121, 2007.

[PST07] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, 2007.

[PSW92] T. Pavlides, J. Swartz, and Y. Wang. Information encoding with two dimensional barcodes. *IEEE Computer*, 25:18–28, 1992.

[PT05] A. Pothen and S. Toledo. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 59:1–59:29. Chapman & Hall/CRC Press, 2005.

[PTUW11] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research.microsoft.com*, 2011.

[Pug86] G. Allen Pugh. Partitioning for selective assembly. *Computers and Industrial Engineering*, 11:175–179, 1986.

[PV96] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations. *Discrete & Computational Geometry*, 16:419–543, 1996.

[Rab80] M. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12:128–138, 1980.

[Ram05] R. Raman. Data structures for sets. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 33:1–33:22. Chapman & Hall/CRC Press, 2005.

[Raw92] G. Rawlins. *Compared to What?* Computer Science Press, New York, 1992.

[RBT04] H. Romero, C. Brizuela, and A. Tchernykh. An experimental comparison of approximation algorithms for the shortest common superstring problem. In *Proc. Fifth Mexican Int. Conf. In Computer Science (ENC'04)*, pages 27–34, 2004.

[RC55] Rand-Corporation. *A Million Random Digits with 100,000 Normal Deviates*. The Free Press, 1955.

[RD18] Edward M. Reingold and Nachum Dershowitz. *Calendrical Calculations: The Ultimate Edition*. Cambridge University Press, 2018.

[RDC93] E. Reingold, N. Dershowitz, and S. Clamen. Calendrical calculations II: Three historical calendars. *Software — Practice and Experience*, 22:383–404, 1993.

[Rei72] E. Reingold. On the optimality of some set algorithms. *J. ACM*, 19:649–659, 1972.

[Rei91] G. Reinelt. TSPLIB–a traveling salesman problem library. *ORSA J. Computing*, 3:376–384, 1991.

[Rei94] G. Reinelt. The traveling salesman problem: Computational solutions for TSP applications. In *Lecture Notes in Computer Science 840*, pages 172–186. Springer-Verlag, 1994.

[RF06] S. Roger and T. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett, 2006.

[RFS98] M. Resende, T. Feo, and S. Smith. Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP. *ACM Transactions on Mathematical Software*, 24:386–394, 1998.

[RHG07] S. Richter, M. Helert, and C. Gretton. A stochastic local search approach to vertex cover. In *Proc. 30th German Conf. on Artificial Intelligence (KI-2007)*, 2007.

[RHS89] A. Robison, B. Hafner, and S. Skiena. Eight pieces cannot cover a chessboard. *Computer Journal*, 32:567–570, 1989.

[Riv92] R. Rivest. The MD5 message digest algorithm. RFC 1321, 1992.

[Rou17] T. Roughgarden. *Algorithms Illuminated*, volume 1. Sound like yourself Publishing, 2017.

[RR99] C. C. Ribeiro and M. G. C. Resende. Algorithm 797: Fortran subroutines for approximate solution of graph planarization problems using GRASP. *ACM Transactions on Mathematical Software*, 25:341–352, 1999.

[RS96] H. Rau and S. Skiena. Dialing for documents: an experiment in information theory. *Journal of Visual Languages and Computing*, pages 79–95, 1996.

[RSA78] R. Rivest, A. Shamir, and L. Adleman: A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[RSL77] D. Rosenkrantz, R. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Computing*, 6:563–581, 1977.

[RSS02] E. Rafalin, D. Souvaine, and I. Streinu. Topological sweep in degenerate cases. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 273–295, 2002.

[RSST96] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. Efficiently fourcoloring planar graphs. In *Proc. 28th ACM Symp. Theory of Computing*, pages 571–575, 1996.

[Rus03] F. Ruskey. Combinatorial Generation. Preliminary working draft. University of Victoria, Victoria, BC, Canada. Draft available at http: //www-csc.uvic.ca/home/fruskey/cgi-bin/html/main.html, 2003.

[RW09] Frank Ruskey and Aaron Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, 2009.

[RW18] Sharath Raghvendra and Mariëtte C Wessels. A grid-based approximation algorithm for the minimum weight triangulation problem. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 101–120. SIAM, 2018.

[Ryt85] W. Rytter. Fast recognition of pushdown automata and context-free languages. *Information and Control*, 67:12–22, 1985.

[RZ05] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. *Tighter Bounds for Graph Steiner Tree Approximation*, pages 122–134, 2005.

[SA95] M. Sharir and P. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, 1995.

[Sah05] S. Sahni. Double-ended priority queues. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 8:1–8:23. Chapman & Hall/CRC Press, 2005.

[Sal06] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, fourth edition, 2006.

[Sam05] H. Samet. Multidimensional spatial data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 16:1–16:29. Chapman & Hall/CRC Press, 2005.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[San00] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[Sav97] C. Savage. A survey of combinatorial gray codes. *SIAM Review*, 39:605–629, 1997.

[Sax80] J.B̃. Saxe. Dynamic programming algorithms for recognizing smallbandwidth graphs in polynomial time. *SIAM J. Algebraic and Discrete Methods*, 1:363–369, 1980.

[Say17] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, fifth edition, 2017.

[SB01] A. Samorodnitsky and A. Barvinok. The distance approach to approximate combinatorial counting. *Geometric and Functional Analysis*, 11:871–899, 2001.

[SB19] Yihan Sun and Guy E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173. SIAM, 2019.

[SBdB16] Punam K. Saha, Gunilla Borgefors, and Gabriella Sanniti di Baja. A survey on skeletonization algorithms and their applications. *Pattern Recognition Letters*, 76:3–12, 2016.

[SBW17] Sima Soltanpour, Boubakeur Boufama, and Q.M. Jonathan Wu. A survey of local feature methods for 3d face recognition. *Pattern Recognition*, 72:391–406, 2017.

[Sch98] A. Schrijver. Bipartite edge-coloring in $O(\delta m)$ time. *SIAM J. Computing*, 28:841–846, 1998.

[Sch11] Hans-Jorg Schulz. Treevis.net: A tree visualization reference. *IEEE Computer Graphics and Applications*, 31(6):11–15, 2011.

[Sch15] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, twentieth anniversary edition, 2015.

[SD75] M. Syslo and J. Dzikiewicz. Computational experiences with some transitive closure algorithms. *Computing*, 15:33–39, 1975.

[SD76] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23:433–445, 1976.

[SDC16] Jonathan Shewchuk, Tamal K Dey, and Siu-Wing Cheng. *Delaunay mesh generation*. Chapman & Hall/CRC Press, 2016.

[SDK83] M. Syslo, N. Deo, and J. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall, 1983.

[Sed77] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9:137–164, 1977.

[Sed78] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21:847–857, 1978.

[Sed98] R. Sedgewick. *Algorithms in C++, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley, third edition, 1998.

[Sei18] R. Seidel. Convex hull computations. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 687–704. CRC Press, 2018.

[SFG82] M. Shore, L. Foulds, and P. Gibbons. An algorithm for the Steiner problem in graphs. *Networks*, 12:323–333, 1982.

[SH75] M. Shamos and D. Hoey. Closest point problems. In *Proc. Sixteenth IEEE Symp. Foundations of Computer Science*, pages 151–162, 1975.

[SH99] W. Shih and W. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[Sha87] M. Sharir. Efficient algorithms for planning purely translational collisionfree motion in two and three dimensions. In *Proc. IEEE Internat. Conf. Robot. Autom.*, pages 1326–1331, 1987.

[She97] J. Ř. Shewchuk. Robust adaptive floating-point geometric predicates.*Disc. Computational Geometry*, 18:305–363, 1997.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficientsorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[Sho09] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, second edition, 2009.

[Si15] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):11, 2015.

[Sin12] Alistair Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Springer Science & Business Media, 2012.

[Sip05] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[SK86] T. Saaty and P. Kainen. *The Four-Color Problem*. Dover, New York, 1986

[SK99] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. CSLIPublications, Stanford University, 1999.

[SK00] R. Skeel and J. Keiper. *Elementary Numerical Computing with Mathematica*. Stipes Pub Llc., 2000.

[Ski88] S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:775–784, 1988.

[Ski90] S. Skiena. *Implementing Discrete Mathematics*. Addison-Wesley, 1990.

[Ski99] S. Skiena. Who is interested in algorithms and why?: lessons from the stony brook algorithms repository. *ACM SIGACT News*, pages 65–74, September 1999.

[Ski12] S. Skiena. Redesigning viral genomes. *IEEE Computer*, 45:47–53, March 2012.

[SL07] M. Singh and L. Lau. Approximating minimum bounded degree spanning tree to within one of optimal. In *Proc. 39th Symp. Theory Computing(STOC)*, pages 661–670, 2007.

[SLL02] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: UserGuide and Reference Manual*. Addison Wesley, 2002.

[SLW+12] Y. Song, Y. Liu, C. Ward, S. Mueller, B. Futcher, S. Skiena, A. Paul, and E. Wimmer. Identification of two functionally redundant RNA elements in the coding sequence of poliovirus using computer-generated design. *Proc. National Academy of Sciences*, 109(36):14301–14307, 2012.

[SM73] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. Fifth ACM Symp. Theory of Computing*, pages 1–9, 1973.

[SM10] David Salomon and Giovanni Motta. *Handbook of Data Compression*. Springer Science & Business Media, 2010.

[ŞMK12] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open MotionPlanning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012. http://ompl.kavrakilab.org.

[Sno18] J. Snoeyink. Point location. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1005–1028. CRC Press, 2018.

[SP08] Kaleem Siddiqi and Stephen Pizer. *Medial Representations: Mathematics, Algorithms and Applications*, volume 37. Springer Science & Business Media, 2008.

[SP15] Jared T. Simpson and Mihai Pop. The theory and practice of genome sequence assembly. *Annual Review of Genomics and Human Genetics*, 16:153–172, 2015.

[SR83] K. Supowit and E. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1983.

[SR03] S. Skiena and M. Revilla. *Programming Challenges: The Programming Contest Training Manual*. Springer-Verlag, 2003.

[SRM14] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559. IEEE, 2014.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[SS07] K. Schurmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37:309–329, 2007.

[SSTF12] Michael Stiebitz, Diego Scheide, Bjarne Toft, and Lene M Favrholdt. *Graph Edge Coloring: Vizing's Theorem and Goldberg's Conjecture*, volume 75. John Wiley & Sons, 2012.

[ST04] D. Spielman and S. Teng. Smoothed analysis: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51:385–463, 2004.

[Sta06] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, fourth edition, 2006.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[STV17] Ola Svensson, Jakub Tarnawski, and László A Végh. A constant-factor approximation algorithm for the asymmetric traveling salesman problem. *arXiv preprint arXiv:1708.04215*, 2017.

[SV87] J. Stasko and J. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

[SW86] Q. Stout and B. Warren. Tree rebalancing in optimal time and space. *Comm. ACM*, 29:902–908, 1986.

[SW11] R. Sedgewick and K. Wayne. *Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley Professional, fourth edition, 2011.

[SW13] Maxim Sviridenko and Justin Ward. Large neighborhood local search for the maximum set packing problem. In *International Colloquium on Automata, Languages, and Programming*, pages 792–803. Springer, 2013.

[SWA03] S. Schlieimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. ACM SIGMOD Int. Conf. on Management of data*, pages 76–85, 2003.

[SWM95] J. Shallit, H. Williams, and F. Moraine. Discovery of a lost factoring machine. *The Mathematical Intelligencer*, 17-3:41–47, Summer 1995.

[SY18] V. Sharma and C. Yap. Robust geometric computation. In J. Goodman, J. O'Rourke, and C. Toth, editors, *Handbook of Discrete and Computational Geometry*, pages 1189–1224. CRC Press, 2018.

[Szp03] G. Szpiro. *Kepler's Conjecture: How Some of the Greatest Minds in History Helped Solve One of the Oldest Math Problems in the World*. John Wiley & Sons, 2003.

[TABN16] Krisnaiyan Thulasiraman, Subramanian Arumugam, Andreas Brandstädt, and Takao Nishizeki. *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*. Chapman & Hall/CRC Press, 2016.

[Tam13] Roberto Tamassia. *Handbook of Graph Drawing and Visualization*. Chapman & Hall/CRC Press, 2013.

[Tar95] G. Tarry. Le problème de labyrinthes. *Nouvelles Ann. de Math.*, 14:187, 1895.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[Tar75] R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[Tar79] R. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Computer and System Sciences*, 18:110–127, 1979.

[Tar83] R. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[TDS+16] Andrea Tagliasacchi, Thomas Delame, Michela Spagnuolo, Nina Amenta, and Alexandru Telea. 3d skeletons: A state-of-the-art report. In *Computer Graphics Forum*, volume 35, pages 573–597. Wiley Online Library, 2016.

[TH03] R. Tam and W. Heidrich. Shape simplification based on the medial axistrans form. In *Proc. 14th IEEE Visualization (VIS-03)*, pages 481–488, 2003.

[THG94] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–80, 1994.

[Thi03] H. Thimbleby. The directed Chinese postman problem. *Software Practice and Experience,*, 33:1081–1096, 2003.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[Tin90] G. Tinhofer. Generating graphs uniformly at random. *Computing*, 7:235–255, 1990.

[TOG18] C. Toth, J. O'Rourke, and J. Goodman, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, third edition, 2018.

[Tro62] H. F. Trotter. Perm (algorithm 115). *Comm. ACM*, 5:434–435, 1962.

[Tur88] J. Turner. Almost all $k$-colorable graphs are easy to color. *J. Algorithms*, 9:63–82, 1988.

[TV01] R. Tamassia and L. Vismara. A case study in algorithm engineering for geometric computing. *Int. J. Computational Geometry and Applications*, 11(1):15–70, 2001.

[TV14] Paolo Toth and Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*. SIAM, 2014.

[TW88] R. Tarjan and C. Van Wyk. An O($n$ lg lg $n$) algorithm for triangulating a simple polygon. *SIAM J. Computing*, 17:143–178, 1988.

[Ukk92] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Intern. Federation of Information Processing (IFIP '92)*, pages 484–492, 1992.

[Val79] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val02] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002

[Van98] B. Vandegriend. Finding hamiltonian cycles: Algorithms, graphs and performance. M. S. Thesis, Dept. of Computer Science, Univ. of Alberta, 1998.

[Vaz04] V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[VB96] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zulstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vel01] Remco C Veltkamp. Shape matching: Similarity measures and algorithms. In *Proceedings International Conference on Shape Modeling and Applications*, pages 188–197. IEEE, 2001.

[Vit01] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.

[Viz64] V. Vizing. On an estimate of the chromatic class of a $p$-graph (in Russian). *Diskret. Analiz*, 3:23–30, 1964.

[vL90a] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A, pages 525–631. MIT Press, 1990.

[vL90b] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Algorithms and Complexity*, volume A. MIT Press, 1990.

[VL05] Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *International Computing and Combinatorics Conference*, pages 440–449. Springer, 2005.

[Vos92] S. Voss. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics*, 40:45 – 72, 1992.

[War62] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, 1962.

[Wat03] B. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Science of Computer Programming*, 48:81–97, 2003.

[Wat04] D. Watts. *Six Degrees: The Science of a Connected Age*. W.W. Norton, 2004.

[WC04] B. Wu and K. Chao. *Spanning Trees and Optimization Problems*. Chapman-Hall/CRC Press, 2004.

[WCL+14] Thomas Weise, Raymond Chiong, Jorg Lassig, Ke Tang, Shigeyoshi Tsutsui, Wenxiang Chen, Zbigniew Michalewicz, and Xin Yao. Benchmarking optimization algorithms: An open source framework for the traveling salesman problem. *IEEE Computational Intelligence Magazine*, 9(3):40–52, 2014.

[Wei73] P. Weiner. Linear pattern-matching algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[Wei11] M. Weiss. *Data Structures and Algorithm Analysis in Java*. Pearson, third edition, 2011.

[Wel84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17–6:8–19, 1984.

[Wel13] René Weller. *New Geometric Data Structures for Collision Detection and Haptics*. Springer Science & Business Media, 2013.

[Wes89] Jeffery R Westbrook. Algorithms and data structures for dynamic graph problems. Princeton University, 1989.

[Wes00] D. West. *Introduction to Graph Theory*. Prentice-Hall, second edition, 2000.

[WF74] R. A.Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

[WH15] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[Whi32] H. Whitney. Congruent graphs and the connectivity of graphs. *American J. Mathematics*, 54:150–168, 1932.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.

[WHS07] Gerhard Wäscher, Heike Hausner, and Holger Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[Wig83] A. Wigerson. Improving the performance guarantee for approximate graph coloring. *J. ACM*, 30:729–735, 1983.

[Wil64] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[Wil84] H. Wilf. Backtrack: An O(1) expected time algorithm for graph coloring. *Info. Proc. Letters*, 18:119–121, 1984.

[Wil85] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Computing*, 14:232–253, 1985.

[Wil89] H. Wilf. *Combinatorial Algorithms: An Update*. SIAM, 1989.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, pages 887–898. ACM, 2012.

[Wil19] D. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.

[Win68] S. Winograd. A new algorithm for inner product. *IEEE Trans./ Computers*, C-17:693–694, 1968.

[WM92a] S. Wu and U. Manber. Agrep — a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.

[WM92b] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35:83–91, 1992.

[Woe03] G. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization — Eureka! You Shrink!*, volume 2570 Springer-Verlag LNCS, pages 185–207, 2003.

[Wol79] T. Wolfe. *The Right Stuff*. Bantam Books, Toronto, 1979.

[WS11] David P.Williamson and David B Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[WSR13] Kai Wang, Steven Skiena, and Thomas G. Robertazzi. Phase balancing algorithms. *Electric Power Systems Research*, 96:218–224, 2013.

[WSSJ14] JingdongWang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

[WW95] F. Wagner and A. Wolff. Map labeling heuristics: provably good and practically useful. In *Proc. 11th ACM Symp. Computational Geometry*, pages 109–118, 1995

[WY05] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT, LNCS v. 3494*, pages 19–35, 2005.

[XWW11] Zhong Xie, Huimin Wang, and Liang Wu. The improved Douglas-Peucker algorithm based on the contour character. In *2011 19th International Conference on Geoinformatics*, pages 1–5. IEEE, 2011.

[Yan03] S. Yan. *Primality Testing and Integer Factorization in Public-Key Cryptography*. Springer, 2003.

[Yao81] A. C. Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, 1981.

[YLCZ05] R. Yeung, S-Y. Li, N. Cai, and Z. Zhang. *Network Coding Theory*. http://www.nowpublishers.com/, Now Publishers, 2005.

[YLDF16] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.

[YM08] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.

[You67] D. Younger. Recognition and parsing of context-free languages in time $O(n^3)$. *Information and Control*, 10:189–208, 1967.

[YS96] F. Younas and S. Skiena. Randomized algorithms for identifying minimal lottery ticket sets. *Journal of Undergraduate Research*, 2–2:88–97, 1996.

[YZ99] E. Yang and Z. Zhang. The shortest common superstring problem: Average case analysis for both exact and approximate matching. *IEEE Trans./ Information Theory*, 45:1867–1886, 1999.

[ZL78] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans./ Information Theory*, IT-23:337–343, 1978.

[ZS04] Z. Zaritsky and M. Sipper. The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm. *IEEE Trans./ Evolutionary Computation*, 8:443–455, 2004.

[Zwi01] U. Zwick. Exact and approximate distances in graphs — a survey. In *Proc. 9th Euro. Symp. Algorithms (ESA)*, pages 33–48, 2001.

ПРИЛОЖЕНИЕ

# Описание электронного архива

Электронный архив, сопровождающий книгу, можно скачать с FTP-сервера издательства «БХВ» по адресу: **https://zip.bhv.ru/9785977567992.zip**. Ссылка на него доступна и со страницы книги на сайте **https://bhv.ru/**.

В архиве содержатся:

- файл Рисунки.pdf, содержащий все рисунки книги в том виде, как они приведены в полноцветном исходном ее издании (с переводом на русский язык некоторых нуждающихся в этом надписей);
- файл Листинги с цветными элементами.pdf, содержащий все листинги книги, содержащие цветные элементы, в том виде, как они приведены в полноцветном исходном ее издании.

# Предметный указатель


## A

agrep 765

## B

BioJava 510
Boost Graph Library, библиотека 514
Boost, библиотека 787
BSP-дерево 521
B-дерево 501

## C

Calendrical 601
CALGO 788
Chaco 674
Cliquer 657
CLP, библиотека 546
Cocone 730
Combinatorica 244, 789
Core, библиотека 697
Crypto++, библиотека 775
Структуры данных 95
Суффиксные деревья 497

## D

DSATUR 677

## F

FFTPACK, библиотека 567
FFTW, библиотека 567
FLUTE 688

## G

GEOMPACK 733
GeoSteiner 688
GLPK 546
GMP, библиотека 559
GnuPG 775
GOBLIN, библиотека 640
Grail+ 778
gzip 769

## I

ILOG CP 604

## J

JAMA, библиотека 537
JC, библиотека 502
JFKAP 778
JGraphT, библиотека 514, 613
JOBSHOP 604
JScience, библиотека 537
JUNG, библиотека 514, 613

## K

KDTREE 2 523
Kd-деревья 497, 520

## L

LEKIN 604
LIBSVM, библиотека 740
LINPACK 537
lp_solve 545

## M

Mathematica 244
METIS 674
MINCUTLIB 640
MIRACL, библиотека 554

## N

NEOS 542, 546
Netlib, библиотека 788
Nettle, библиотека 775
NTL, библиотека 554

## P

PARI 554
PAUP 688
PHYLIP 688
PIGALE 654
Powercrust 730, 737
Prolog, язык 390

## Q

Qhull 701, 705, 709
QSlim 736

## R

RAM 55
RAM-машина 55
RAM-модель 55
REDUCE 519
Roncorde 667
Ронтейнеры 95

## S

Scotch 674
Soundex, алгоритм 765
SourceForge 788
Spatial Index Demos 523
SPRNG, библиотека 551
Stanford GraphBase 789
STL, библиотека 502
strmat 510, 760
SYMPHONY 754, 757

## T

TRE, библиотека 765
TSPLIB, библиотека 667

## V

VRONI 730

## А

Ависа — Фукуды, алгоритм 701
Автоморфизм 681
Аксиома треугольника 445, 665
Алгоритм 25
◊ Soundex 765
◊ Ависа — Фукуды 701
◊ Ахо — Корасика 759
◊ Беллмана — Форда 624
◊ ближайшего соседа 27
◊ ближайших пар 28
◊ Бойера — Мура 759
◊ Борувки 619
◊ генетический 471
◊ Гровера 475
◊ Грэхема 700
◊ Дейкстры 299, 301, 623
◊ Дугласа — Пекера 736
◊ заворачивания подарка 699
◊ заметающей прямой 705
◊ Карацубы 191
◊ Кнута — Морриса — Пратта 759
◊ Кристофидеса 446, 447
◊ Крускала 288, 618
◊ Крускала 290
◊ Лас-Вегас 207
◊ Лемпеля — Зива 769
◊ Монте-Карло 207, 227, 316
◊ поиска подстроки в строке 225
◊ поразрядной сортировки (radix sort) 172
◊ Прима 285, 286, 300, 618
◊ Рабина — Карпа 225, 315
◊ «разделяй и властвуй» 194
◊ сортировки методом Шелла (shellsort) 172
◊ сортировки слиянием 172, 181
◊ тасования Фишера — Йейтса 585
◊ Укконена 510
◊ Флойда — Уоршелла 302, 304, 625
◊ Форчуна 707
◊ Хиршберга 764
◊ Шора 478
◊ Эдмондса — Карпа 314
Альфа-очертания 700
Анаграмма 349
Аналогии 49
Арифметическая прогрессия 72
Арифметические операции 557
Асимптотические обозначения 58, 59, 61
Асимптотическая нотация 58

## Б

Базовые объекты 44
Бесконтурные графы 612
Беспорядок (derangement) 347
Библиотека
◊ Boost 787
◊ Boost Graph Library 514
◊ CLP 546
◊ Core 697
◊ Crypto++ 775
◊ FFTPACK 567
◊ FFTW 567
◊ GMP 559
◊ GOBLIN 640
◊ JAMA 537
◊ JC 502
◊ JGraphT 514, 613
◊ JScience 537
◊ JUNG 514, 613
◊ LIBSVM 740
◊ MIRACL 554
◊ Netlib 788
◊ Nettle 775
◊ NTL 554
◊ SPRNG 551
◊ STL 502
◊ TRE 765
◊ TSPLIB 667
Биномиальные коэффициенты 358
Биноминальное распределение 212
Блочная сортировка (bucket sort) 168
Борувки, алгоритм 619
Быстрая сортировка (quicksort) 138
Быстрое преобразование Фурье 476

## В

Вершинная раскраска 675
Вершинное покрытие 410, 412, 419, 441, 661
Взвешенные графы (weighted graph) 283
Взятие адреса 98
Возведение в степень 75
Вороного, диаграмма 706
Восстановление
◊ пирамиды (heapify) 150
◊ пути 364
Восхождение по выпуклой поверхности 456
Выпуклая оболочка 140, 409, 698
Выпуклые функции 540
Вырожденность 694
Вырожденные системы уравнений 527
Высота двоичных деревьев поиска 112

## Г

Гамильтонов путь 117
Гамильтонов цикл 410, 423, 668
Гармоническое число 76
Генератор
◊ псевдослучайных чисел 230
◊ случайных чисел 229, 547
Генетические алгоритмы 471
Геометрическая прогрессия 72
Геометрическое распределение 216
Гиперграф 513
Грамматика контекстно-свободная 384
Граничные ошибки 39
Граф 41, 233, 234, 595
◊ ациклический 236
◊ бесконтурный подграф 449
◊ вершинная раскраска 675
◊ вершинное покрытие 278, 412, 419, 441, 661
◊ взвешенный 235, 238, 283
◊ гамильтонов цикл 410, 668
◊ гиперграф 513
◊ двудольный 258, 310
◊ доминирующее множество 662
◊ изоморфизм 680
◊ квадрат 278
◊ клика 415
◊ кратное ребро 235
◊ кратчайший путь 622
◊ матрица инцидентности 513
◊ минимальное остовное дерево 617
◊ мост 268
◊ независимое множество 413, 463
◊ неориентированный 237
◊ неявный 236
◊ обход 250
◊ ориентированное дерево 281
◊ ориентированный 234, 237
◊ остаточного потока (residual flow graph) 311
◊ остовное дерево 284
◊ паросочетание 310
◊ перечисление путей 330
◊ петля 235
◊ планарный 512

Граф *(прод.)*
◊ плотный 235, 238
◊ полный 236
◊ помеченный 237
◊ простой 238
◊ путь 255
◊ разреженный 235
◊ разрывающее множество 689
◊ раскраска вершин 257
◊ реберное покрытие 323, 663
◊ рисование 644
◊ связность 264, 610, 638
◊ связный 256
◊ сильно связный 272
◊ список смежности 283
◊ степень вершины 238
◊ топологическая сортировка 236, 270
◊ транзитивное замыкание 304
◊ турнир 281
◊ укладка 236
◊ цикл 263
◊ шарнир 264
◊ Эйлеров подграф 436
◊ Эйлеров цикл 436
График охлаждения 460
Грея, код 588
Грэхема, алгоритм 700

## Д

Двоичное дерево 74
Двоичные пирамиды 505
Двоичный поиск 73, 181, 576
Дейкстры, алгоритм 299, 623
Демонстрация доказательства 33
Дерево 41, 74
◊ BSP-дерево 521
◊ B-дерево 501
◊ kd-дерево 520
◊ вставка элемента 111
◊ двоичное 74, 108
◊ квадродерево 521
◊ косое 501
◊ минимальное остовное 617
◊ наибольший элемент 110
◊ обход 110
◊ октадерево 521
◊ остовное 284
◊ поиска 108
◊ помеченное 597
◊ рисование 649
◊ сбалансированное 113
◊ суффиксное 507
◊ удаление элемента 112
◊ Штейнера 294, 617, 685
Деревья 236
Диаграмма
◊ Вороного 706
◊ Ганта 605
◊ Феррера 593
Диапазонный запрос 714
Динамическое
◊ выделение памяти 96, 101
◊ программирование 181, 352
Дискретное преобразование Фурье 565
Доказательство
◊ алгоритма 33
◊ сложности 423
Доминирование функции 63, 84
Доминирующее множество 662
Дополнение графа 659
Древесное ребро 443
Дугласа — Пекера, алгоритм 736

## З

Задача
◊ 3-SAT 417, 426
◊ безусловной оптимизации 575
◊ вершинной раскраски 675
◊ внешней сортировки 572
◊ выполнимости 605
  ▫ булевых формул 416, 605
◊ выявления изоморфизма графов 680
◊ выявления сходства фигур 737
◊ календарного планирования 30, 413, 601
◊ китайского почтальона 634
◊ коммивояжера 30, 352, 387, 664
◊ линейного расположения 531
◊ максимального разреза 463
◊ минимизации профиля 531
◊ нечеткого сравнения строк 360, 761
◊ о вершинном покрытии 412, 419, 441, 661
◊ о доминирующем множестве 662
◊ о клике 415
◊ о минимальном множестве представителей 752
◊ о назначениях 631
◊ о независимом множестве 413

◊ о покрытии множества 750
◊ о потоках в сетях 309, 641
◊ о разрывающем множестве 689
◊ о реберном покрытии 663
◊ о рюкзаке 376, 560
◊ о сумме подмножества 562
◊ оптимизации 538
◊ планирования перемещений 741
◊ поиска ближайшего соседа 710
◊ поиска ближайшей пары 406
◊ поиска в области 714
◊ поиска гамильтонова цикла 410, 668
◊ поиска компонентов связности 610
◊ поиска кратчайшего пути 622
◊ поиска медианы 580
◊ поиска эйлерова цикла 634
◊ построения выпуклой оболочки 698
◊ построения дерева Штейнера 685
◊ преобразования к срединной оси 728
◊ разбиения графа 672
◊ разбиения многоугольника на части 732
◊ разбиения множества целых чисел 562
◊ раскроя 725
◊ реберной раскраски 679
◊ рисования графа 644
◊ рисования дерева 649
◊ сортировки 25, 56
◊ сравнения строк 758
◊ укладки множества 755
◊ упорядоченного разбиения (ordered partition) 381
◊ упрощения многоугольников 735
Задачи разрешимости (decision problems 405
Закон
◊ Кирхгофа 527
◊ Мура 246
◊ Ципфа 576
Запоминание (сохранение) результатов (memoization) 355

## И

Идентичность графов 681
Изоморфизм графов 680
Имитация отжига 459
Индекс 96
Интерполяционный поиск 578
Использование
◊ «жадного» эвристического алгоритма 118
◊ циклического избыточного кода 773

## К

Календарное планирование 30, 601
Календарь 599
Каноникализация 125
Каркас транзитивного замыкания 630
Квадратный корень 184
Квадродерево 521
Классы
◊ абстрактных типов данных 95
◊ ребер неориентированного графа 261
Клика 656
Код
◊ Грея 588, 669
◊ Прюфера 597
◊ Хаффмана 768
Коллизия 121, 122
Компонент связности 256, 610
Конечный автомат 776
Контейнер 101
Контекстно-свободная грамматика 384
Контрольная сумма 773
Контрпримеры 35
Конфигурации прямых 745
Конфликт имен файлов 318
Косое дерево 501
Коэффициент сходства Жаккара 222
Красно-черные деревья 501
Кратчайший путь 622
Криптографическое хеширование 124
Криптография 770
Крускала, алгоритм 288, 618
Кэширование 354
◊ результатов вычислений 356

## Л

Лагранжева релаксация 541
Лексикографический порядок 583, 588
Лемпеля — Зива, алгоритм 769
Ленточная матрица 528
Линейное программирование 542
Линейное расширение 614
Линейный конгруэнтный генератор 229, 549
Логарифм 73
◊ двоичный 78
◊ десятичный 78
◊ натуральный 78
Локальный поиск 455

## М

Максимально монотонная подпоследовательность 370
Максимальное независимое множество 660
Малая теорема Ферма 227
Массив 96
◊ разбиение на части 161
Математическая индукция 38
Матрица смежности 240, 511
Матрицы: умножение 70
Матроиды (matroids) 621
Машина с произвольным доступом к памяти (Random Access Machine, RAM) 55
Медиана 579, 581
Местоположение
◊ точка 696, 716
◊ точки на плоскости 716
Метод
◊ внутренней точки 544
◊ восхождения по выпуклой поверхности 456
◊ деления интервала пополам 184
◊ доказательства от противного 44
◊ Дэвиса — Путнама 606
◊ идеального хеширования 220
◊ имитации отжига 459
◊ инкрементального изменения 583
◊ исключения Гаусса 527
◊ «лесного пожара» 730
◊ минимальных хеш-кодов (minwise hashing) 223
◊ Монте-Карло 452
◊ отпечатков пальцев 126
◊ поиска «лучший-первый» (best-first) 343
◊ полос 719
◊ произвольной выборки 452
◊ «разделяй и властвуй» 181
◊ ранжирования/деранжирования 583
◊ табличного сохранения, tabling 355
Методические вычисления 49
Методы
◊ машинного обучения 739
◊ произвольной выборки 206
◊ Якоби и Гаусса — Зейделя 528
Механизм обмена цветов 677
Минимальное остовное дерево 284, 617
Минковского, сумма 748
Многоугольник 42
◊ разбиение на части 731
Множество 515, 750
◊ пересечение 141
◊ разбиение 517
Мода 581
Моделирование задачи 40
Модель
◊ вычислений RAM 55
◊ Эрдёша — Реньи 596
Мост 638
Мультимножество 585

## Н

Нагруженное дерево 507
Наибольший общий делитель 408
Наименьшее общее кратное 408
Невзвешенные графы (unweighted graphs) 283
Независимое множество 659
◊ вершин графа 413
Неправильность алгоритма 35
Нормальная форма Хомского 385

## О

Обход
◊ в глубину (depth-first search, DFS) 325
◊ в ширину (breadth-first search, BFS) 325
Обход графа 250
◊ в глубину 259
◊ в ширину 251
Односторонний двоичный поиск 183, 577
Октадерево 521
Операция
◊ дополнения графа 415
◊ стягивания 315
Оптимальное дерево двоичного поиска 576
Оптимизация 538
Основная теорема 188
Основные типы графов 234
Остовное дерево 284
Открытая адресация 122
Открытый ключ RSA 552
Отношение доминирования 63, 84
Отсечение (pruning) 332
Очереди с приоритетами 95
Очередь 102
◊ с приоритетами 115, 119, 504

## П

Парадокс дня рождения 220
Параллелизм на уровне данных 194
Паросочетание 310, 323, 447, 630
Перебор с возвратом (backtracking) 324
Пересечение
◊ множеств 141
◊ отрезков 720
◊ прямой и отрезка 696
◊ прямых 694
Перестановка 41, 329, 583
Перманент 536
Петля 238
Пирамида 146
◊ вставка элемента 149
◊ создание 148
◊ удаление элемента 150
Пирамидальная сортировка (heapsort) 138
Пирамидальное число 80
Планирование перемещений 741
Площадь треугольника 696
Подмножество 41, 587
Подстрока 509
Поиск 575
◊ ближайшего соседа 713
◊ ближайшей пары 406
◊ двоичный 108
◊ локальный 455
◊ максимальной общей подстроки 779
◊ минимальной общей надстроки 782
◊ подстроки 509
Полиномиальное умножение 197
Порядок
◊ FIFO 102
◊ LIFO 101
Последовательный поиск 576
Поток в сети 309, 641
Правило Горнера 500
Преобразование к срединной оси 728
Прикидка 49
Прима, алгоритм 285, 618
Примеры свертки 198
Принцип
◊ оптимальности 388
◊ «разделяй и властвуй» 168
Проверка числа на простоту 227, 552
Произвольная выборка 452
Процедура двоичного поиска 183
Прюфера, код 597
Прямолинейный скелет 729
Псевдослучайные числа 548
Пузырьковый метод 150
Путь 255, 298

## Р

Разбиение 590
◊ множества 290, 517
◊ целого числа 590
Разложение
◊ на множители 552
◊ по контейнерам 724
Размещение прямоугольников по корзинам 318
Разреженная матрица 528
Разрывающее множество 689
Разрывающие множества
◊ вершин 615
◊ дуг 615
Разыменование указателя 97
Рандомизация 113, 165, 444
Рандомизированные алгоритмы 58, 206, 217
Расстояние
◊ редактирования 360, 407
◊ хаусдорфово 739
◊ Хемминга 738
Реберное покрытие 663
Реберно-хроматическое число 679
Регулярные выражения 776
Рекуррентное соотношение 186
Рекурсивный алгоритм 181
Рекурсивный объект 42
Рекурсия 38
Решето числового поля 553
Решеточные укладки 653
Рисование
◊ графа 644
◊ дерева 649

## С

Самоорганизующиеся списки 577
Самоорганизующийся список (self-organizing list) 499
Сбалансированное двоичное дерево 113, 146
Сведение (reduction) 403
Свертка 197
Свойства
◊ алгоритма 26
◊ контрпримеров 36

Связность графа 638
Связные структуры данных 95
Селективная сборка 465
Сервер комбинаторных объектов 586, 589
Сжатие текста 766
Сильно связный граф 611
Симметричное распределение 213
Симплекс-метод 543
Синтаксический разбор 384, 385
Система линейных уравнений 527
Слабо связный граф 611
Словари 95
Словарь 102, 498
Сложность алгоритма 56
Случаи основной теоремы 189
Случайные переменные 211
Случайные числа 547
Смежные структуры данных 95
Сортировка 570
◊ блочная 167
◊ быстрая 160
◊ вставками 25, 67, 154
◊ методом выбора 65, 145
◊ пирамидальная 146
◊ порядок 143
◊ распределением (distribution sort) 138, 168
◊ слиянием 158
◊ топологическая 270, 614
Составные события 209
Социальные сети 237
Список 98
◊ вставка элемента 99
◊ поиск элемента 98
◊ смежности 240, 283, 511
◊ удаление элемента 99
Сравнение строк 68, 360, 366, 758, 761
Среднее значение 579
Стек 101
Строка 42, 750
◊ сравнение 68, 360, 366, 761
Строки 225
Структура
◊ с реализацией очереди с приоритетами 146
◊ «поиск-объединение» (union-find) 291
Структуры данных 95
Судоку 333
Сумма Минковского 748
Суффиксное дерево 507
Суффиксный массив 509
Сходство фигур 737

## Т

Текст, сжатие 766
Теорема
◊ Байеса 211
◊ Кука 432
◊ Менгера 639
◊ о зоне 747
Теория вероятностей 207
Типы вершин 251
Топологическая сортировка 236, 614
Точка 42
◊ местоположение 696, 716
Транзитивная редукция 629
Транзитивное замыкание 304, 628
Транспонированный граф 274
Триангуляция 117, 702

## У

Увеличивающие пути 311, 312, 632
Указатели 95, 97
Уменьшение ширины ленты 530
Умножение матриц 70, 532
Уоринга, проблема 79
Уплотнение данных 126
Упорядочивание последовательности 317
Условная вероятность 210
Устойчивые множества 659

## Ф

Фибоначчи 353
Фильтр Блума 218, 219, 517
Флойда — Уоршелла, алгоритм 303, 625
Формула Эйлера 652
Формулы суммирования 71
Функция
◊ накопленных вероятностей (ФНВ) 211
◊ плотности вероятностей (ФПВ) 211
◊ попарного сравнения элементов 144
◊ сравнения 145
◊ хеширования 214, 217
Фурье 565

## Х

Хаусдорфово расстояние 739
Хаффмана, код 768
Хемминга, расстояние 738
Хеширование 123, 126, 131, 142, 214

Хеш-таблица 121, 131
Хеш-функция 121, 500
Хиршберга, алгоритм 764
Хроматический индекс 679

## Ц

Целочисленное программирование 421
Цикл Эйлера 423
Цифровая подпись 774

## Ч

Частично упорядоченное множество 613
Частотное распределение 139
Числа
◊ Кармайкла 227
◊ Фибоначчи 353
Чрезвычайно параллельные (embarrassingly parallel) задачи 194

## Ш

Шарнир 638
◊ графа 264
Шифр
◊ AES 771
◊ DES 771
◊ RSA 772
◊ сдвиг Цезаря 771
Штейнера, дерево 294, 685
Штрихкод 372

## Э

Эвристики 30
Эвристический алгоритм 27
◊ A* 345
◊ приблизительного решения 118
Эйлеров цикл 446, 634
Экземпляр задачи сортировки 25
Эксцентриситет вершины 625

**Стивен С. Скиена**

# Алгоритмы. Руководство по разработке

**3-е издание**

*Перевод с английского*

**Группа подготовки издания:**

| | |
|---|---|
| Руководитель проекта | *Евгений Рыбаков* |
| Зав. редакцией | *Людмила Гауль* |
| Редактор | *Григорий Добин* |
| Компьютерная верстка | *Ольги Сергиенко* |
| Оформление обложки | *Зои Канторович* |

Подписано в печать 05.04.22.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 68,37.
Тираж 2000 экз. Заказ № 318.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ООО "Типография "Миттель Пресс"
127254, г. Москва, ул. Руставели, д. 14, стр. 6.
Тел./факс: +7 (495) 619-08-30, 647-01-89.
E-mail: mittelpress@mail.ru

*Наиболее полное руководство по разработке эффективных алгоритмов*

# Алгоритмы. Руководство по разработке. *Третье издание*

*Это руководство — мой абсолютный фаворит для подготовки к собеседованию. Больше чем любая другая книга она помогла мне понять, насколько поразительно распространены... проблемы с графами — они должны быть частью набора инструментов каждого работающего программиста. В книге также рассматриваются основные структуры данных и алгоритмы сортировки, что является приятным бонусом, а простые и понятные картинки облегчают запоминание.*
**Стив Йегге, публикация «Get that Job at Google»**

*Эта книга сохраняет за собой звание лучшего и наиболее полного практического руководства по алгоритмам... Каждый программист должен ее прочитать и держать под рукой. ...Это лучшая инвестиция... которую может сделать программист.*
**Гарольд Тимблби, журнал «Times Higher Education»**

*Замечательно открыть книгу на любой странице и обнаружить интересный алгоритм. Это единственный учебник, который я храню со студенческих лет!*
**Кори Барт, Делавэрский университет**

Расширенное и обновленное третье издание классического бестселлера продолжает раскрывать «загадку» разработки алгоритмов и анализа их эффективности. Книга является основным учебником для курсов по разработке алгоритмов, пособием для самоподготовки к собеседованиям, сохраняя при этом свой статус главного практического справочника по алгоритмам для программистов, исследователей и студентов.
Первая часть представляет собой общее введение в технические приемы разработки и анализа компьютерных алгоритмов. Вторая часть содержит обширный список литературы и каталог наиболее распространенных алгоритмических задач, для которых перечислены существующие программные реализации.

### Новое в третьем издании

- Расширенный набор рандомизированных алгоритмов, хеширования, алгоритмов «разделяй и властвуй», аппроксимации и квантовых вычислений
- Онлайн-поддержка для преподавателей, включающая слайды и видеоуроки
- Полноцветные иллюстрации и код, наглядно разъясняющие сложные концепции
- Новые «истории из жизни», рассказывающие об опыте работы с реальными приложениями
- Более 100 новых задач, включая задачи по программированию от LeetCode и Hackerrank
- Актуальные ссылки к лучшим реализациям на языках C, C++ и Java

### Дополнительные средства обучения

- Уникальный каталог наиболее часто встречающихся на практике 75 алгоритмических задач с решениями
- Упражнения, решающие типичные проблемы на собеседованиях
- Переработанные и новые домашние задания, закрепляющие основной материал
- Стиль «без доказательств теорем», обеспечивающий доступный и интуитивно понятный подход к сложной теме
- Реальные коды на языке C
- Исчерпывающие ссылки на обзорные статьи и основную литературу

**Стивен С. Скиена** (Steven S. Skiena), профессор кафедры вычислительной техники Университета Стоуни – Брук, известный исследователь алгоритмов, лауреат премии Института IEEE, автор популярной книги «Programming Challenges: The Programming Contest Training Manual» («Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям»).

ISBN 978-5-9775-6799-2

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru



Электронный архив, содержащий цветные рисунки и листинги, можно скачать по ссылке https://zip.bhv.ru/9785977567992.zip, а также со страницы книги на сайте https://bhv.ru