Алгоритмы

с нуля

ПИТЕР

Introduction to Algorithms:

A Comprehensive Guide for Beginners: Unlocking Computational Thinking

Алгоритмы

с нуля

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018
УДК 004.021
К32

Cuantum Technologies

К32 Алгоритмы с нуля. — СПб.: Питер, 2024. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4076-3

Погрузитесь в мир алгоритмов! Разберитесь в их принципах, особенностях проектирования и практического применения.

Вы познакомитесь с различными видами алгоритмов, узнаете их сильные и слабые стороны и поймете, в каких контекстах они лучше всего работают. На практических примерах увидите, как эти мощные инструменты используются для решения задач в информатике, анализе данных, искусственном интеллекте и других областях.

Каждая глава содержит понятные объяснения, наглядные примеры и задачи, помогающие закрепить изученный материал. Особый акцент сделан на вычислительном мышлении и анализе эффективности алгоритмов — важнейших навыках в области современных технологий.

Книга «Алгоритмы с нуля» послужит ценным ресурсом и для новичков, и для профессионалов, желающих отточить свои навыки.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.021

Права на издание получены по соглашению с Cuantum Technologies LLC. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 979-8854326957 англ.
ISBN 978-5-4461-4076-3

© 2023 Cuantum Technologies
© Перевод на русский язык ООО «Прогресс книга», 2024
© Издание на русском языке, оформление ООО «Прогресс книга», 2024
© Серия «Библиотека программиста», 2024

КРАТКОЕ СОДЕРЖАНИЕ

https://t.me/it_boooks/2

КТО МЫ 12

ИСХОДНЫЙ КОД ПРИМЕРОВ 14

ПРЕДИСЛОВИЕ 15

ВВЕДЕНИЕ 17

ГЛАВА 1. ВВЕДЕНИЕ В АЛГОРИТМЫ 22

ГЛАВА 2. ПСЕВДОКОД И БЛОК-СХЕМЫ 47

ГЛАВА 3. ЭФФЕКТИВНОСТЬ АЛГОРИТМОВ 67

ГЛАВА 4. ОСНОВНЫЕ ТИПЫ АЛГОРИТМОВ 86

ГЛАВА 5. АЛГОРИТМЫ ПОИСКА 109

ГЛАВА 6. АЛГОРИТМЫ СОРТИРОВКИ 127

ГЛАВА 7. ГРАФОВЫЕ АЛГОРИТМЫ 156

ГЛАВА 8. СТРУКТУРЫ ДАННЫХ, ИСПОЛЬЗУЕМЫЕ В АЛГОРИТМАХ 181

ГЛАВА 9. МЕТОДЫ ПРОЕКТИРОВАНИЯ АЛГОРИТМОВ 208

ГЛАВА 10. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ АЛГОРИТМОВ 230

ЗАКЛЮЧЕНИЕ 253

ОГЛАВЛЕНИЕ

Кто мы ... 12
Наша философия ... 12
Наш опыт ... 13

Исходный код примеров ... 14

Предисловие ... 15

Введение ... 17
О чем эта книга ... 17
Кому адресована книга ... 18
Как пользоваться изданием ... 19
От издательства ... 21

Глава 1. Введение в алгоритмы ... 22
1.1. Что такое алгоритм ... 22
1.1.1. Характеристики хорошего алгоритма ... 24
1.1.2. Как используются алгоритмы ... 26
1.1.3. Краткое подведение итогов и некоторые моменты для размышления ... 29
1.2. Важность алгоритмов в информатике ... 31
1.2.1. Алгоритмы лежат в основе нашего цифрового мира ... 31
1.2.2. Алгоритмы повышают эффективность ... 32
1.2.3. Алгоритмы лежат в основе передовых технологий ... 33
1.2.4. Будущие последствия и достижения в сфере алгоритмов ... 34
1.3. Основы вычислительного мышления ... 36
1.3.1. Декомпозиция ... 36
1.3.2. Выявление закономерностей ... 37
1.3.3. Абстрагирование ... 37
1.3.4. Алгоритмическое мышление ... 38
1.3.5. Отладка и итерация ... 39
1.4. Практические задачи ... 41
Задача 1. Генератор паролей ... 41
Задача 2. Календарь событий ... 42
Задача 3. Построение пирамиды ... 43
Задача 4. Сжатие текста ... 44
Резюме ... 45

Глава 2. Псевдокод и блок-схемы ... 47
2.1. Псевдокод ... 47
2.1.1. Гибкость псевдокода ... 51
2.2. Блок-схемы ... 52
2.3. Выражение реальных задач в виде псевдокода ... 58
2.4. Практические задачи ... 63
Задача 1. Вывод чисел ... 63
Задача 2. Выявление палиндрома ... 64
Задача 3. Поиск наибольшего числа ... 64
Резюме ... 65

Глава 3. Эффективность алгоритмов ... 67
3.1. Временная сложность ... 67
3.1.1. Нотация «О большое» ... 69
3.1.2. Разница между временной сложностью в лучшем, среднем и худшем случаях ... 70
3.2. Пространственная сложность ... 72
3.2.1. Кэширование/мемоизация ... 74
3.3. Введение в нотацию «О большое» ... 76
3.3.1. Распространенные типы временной сложности ... 77
3.3.2. Асимптотический анализ ... 80
3.4. Практические задачи ... 81
Задача 1. Линейный поиск ... 81
Задача 2. Сумма элементов ... 82
Задача 3. Поиск дубликатов ... 83
Задача 4. Пузырьковая сортировка ... 83
Резюме ... 84

Глава 4. Основные типы алгоритмов ... 86
4.1. Алгоритмы «разделяй и властвуй» ... 86
Рекурсивная природа ... 88
Эффективность ... 89
Потребление памяти ... 89
Параллелизм ... 90
Сортировка слиянием ... 90
Алгоритм Штрассена ... 91
Алгоритм Карацубы ... 91
Ханойские башни ... 92
Пара ближайших точек ... 92
4.2. Жадные алгоритмы ... 93
4.2.1. Что такое жадный алгоритм ... 94
4.2.2. Задача размена монет ... 94
4.3. Алгоритмы динамического программирования ... 99
4.4. Рекурсивные алгоритмы ... 102
4.4.1. Хвостовая рекурсия ... 103

4.5. Практические задачи ... 104
Задача 1. Двоичный поиск (разделяй и властвуй) ... 104
Задача 2. Размен монет (жадный алгоритм) ... 105
Задача 3. Числа Фибоначчи (динамическое программирование) ... 105
Задача 4. Сумма натуральных чисел (рекурсивный алгоритм) ... 106
Задача 5. Быстрая сортировка (разделяй и властвуй) ... 106
Задача 6. Реализация стека с помощью рекурсии (рекурсивный алгоритм) ... 106
Резюме ... 107

Глава 5. Алгоритмы поиска ... 109
5.1. Линейный поиск ... 109
5.1.1. Ограничения линейного поиска ... 113
5.2. Двоичный поиск ... 114
5.3. Хеширование и хеш-таблицы ... 117
5.3.1. Коллизии ... 119
5.4. Практические задачи ... 124
Задача 1. Линейный поиск ... 124
Задача 2. Двоичный поиск ... 124
Задача 3. Хеширование ... 125
Задача 4. Сравнение эффективности двоичного и линейного поиска ... 125
Резюме ... 125

Глава 6. Алгоритмы сортировки ... 127
6.1. Пузырьковая сортировка ... 127
6.1.1. Использование пузырьковой сортировки ... 129
6.2. Сортировка выбором ... 131
6.3. Сортировка вставками ... 134
6.4. Быстрая сортировка ... 136
6.5. Сортировка слиянием ... 142
6.6. Пирамидальная сортировка ... 148
6.7. Практические задачи ... 151
Задача 1. Реализация пузырьковой сортировки ... 151
Задача 2. Анализ худшего случая быстрой сортировки ... 152
Задача 3. Сортировка слиянием связанного списка ... 152
Задача 4. Формирование кучи на основе массива ... 153
Задача 5. Стабильность сортировки ... 153
Резюме ... 154

Глава 7. Графовые алгоритмы ... 156
7.1. Введение в теорию графов ... 156
7.2. Поиск в глубину ... 158
7.3. Поиск в ширину ... 162
7.3.1. Временная сложность BFS ... 164

7.4. Алгоритм Дейкстры ... 166
7.4.1. Взвешенные и невзвешенные графы ... 168
7.4.2. Неотрицательные веса ... 168
7.4.3. Приложения ... 169
7.4.4. Варианты ... 169
7.4.5. Визуализация алгоритма Дейкстры ... 170
7.4.6. Временная сложность ... 170
7.5. Алгоритм поиска A* ... 171
7.5.1. Эвристика в A* ... 172
7.5.2. Временная и пространственная сложности ... 173
7.5.3. Оптимальность алгоритма A* ... 174
7.5.4. Практическое применение алгоритма A* ... 175
7.5.5. Варианты алгоритма A* ... 176
7.5.6. Сложность алгоритма поиска A* ... 176
7.6. Практические задачи ... 177
Задача 1. DFS при поиске путей через лабиринт ... 177
Задача 2. Поиск кратчайшего пути в сетке с помощью BFS ... 178
Резюме ... 179

Глава 8. Структуры данных, используемые в алгоритмах ... 181
8.1. Массивы ... 181
8.1.1. Свойства массивов и основы их применения ... 183
8.2. Связанные списки ... 186
8.2.1. Другие типы связанных списков ... 188
8.3. Стеки и очереди ... 191
8.3.1. Стеки ... 191
8.3.2. Очереди ... 192
8.3.3. Приоритетные и двунаправленные очереди ... 193
8.4. Деревья и графы ... 196
8.4.1. Деревья ... 196
8.4.2. Графы ... 199
8.5. Практические задачи ... 203
Задача 1. Массивы: подмассив с максимальной суммой ... 203
Задача 2. Связанные списки: перестановка элементов списка в обратном порядке ... 204
Задача 3. Стеки: проверка парности скобок ... 204
Задача 4. Деревья: максимальная глубина двоичного дерева ... 205
Резюме ... 206

Глава 9. Методы проектирования алгоритмов ... 208
9.1. Рекурсия ... 208
9.2. Итеративные подходы ... 213
9.2.1. Итеративное вычисление факториала ... 213
9.2.2. Оптимизация хвостовой рекурсии ... 214

9.3. Поиск с возвратом ... 216
9.4. Метод ветвей и границ ... 219
9.4.1. Принцип работы метода ветвей и границ ... 220
9.4.2. Задача о коммивояжере ... 220
9.4.3. Сложность и практическое использование ... 222
9.5. Практические задачи ... 226
Задача 1. Рекурсия: числа Фибоначчи ... 226
Задача 2. Итерации: факториал ... 226
Задача 3. Поиск с возвратом: задача N ферзей ... 226
Задача 4. Ветви и границы: задача о коммивояжере ... 227
Резюме ... 228

Глава 10. Практическое применение алгоритмов ... 230
10.1. Алгоритмы в базах данных ... 230
10.1.1. Двухфазная блокировка (2PL) ... 233
10.1.2. Управление параллельным доступом с помощью многоверсионности ... 233
10.2. Алгоритмы в искусственном интеллекте ... 234
10.2.1. Алгоритмы машинного обучения ... 235
10.2.2. Алгоритмы обработки естественного языка в ИИ ... 240
10.2.3. Роль алгоритмов в машинном обучении ... 242
10.3. Алгоритмы сетевой маршрутизации ... 245
10.3.1. Алгоритм Дейкстры ... 245
10.3.2. Алгоритм Беллмана — Форда ... 246
10.3.3. Протокол маршрутизации по состоянию канала ... 247
10.4. Практические задачи ... 248
Задача 1. Алгоритмы в базах данных ... 248
Задача 2. Алгоритмы искусственного интеллекта ... 249
Задача 3. Алгоритмы сетевой маршрутизации ... 250
Резюме ... 251

Заключение ... 253
Дальнейшие действия ... 254

Искусственный интеллект, глубокое обучение,
машинное обучение — чем бы вы ни занимались,
если вы чего-то не понимаете в этих темах, то потратьте
время на их изучение. В противном случае через
три года вы превратитесь в динозавра.

Марк Кьюбан, предприниматель и инвестор

КТО МЫ

Cuantum Technologies — ведущая инновационная компания в сфере разработки программного обеспечения (ПО) и образования, уделяющая особое внимание использованию возможностей искусственного интеллекта и передовых технологий.

Мы специализируемся на разработке ПО для веб-приложений, написании литературы по программированию и искусственному интеллекту, а также на создании привлекательных веб-приложений с использованием HTML, CSS, JavaScript и Three.js. В наш разнообразный ассортимент продуктов входят CuantumAI — инновационное предложение SaaS (software as a service — программное обеспечение как услуга) — и множество книг, посвященных Python, NLP, PHP, JavaScript и многому другому.

Наша философия

Цель Cuantum Technologies — в разработке инструментов, которые позволяют людям улучшать свою жизнь с помощью искусственного интеллекта и новых технологий. Мы верим, что технологии — не просто инструмент, а средство, способствующее позитивным изменениям и развитию всех аспектов нашей жизни.

Мы стремимся не только к технологическому прогрессу, но и к формированию будущего, в котором каждый человек будет иметь доступ к знаниям и инструментам, позволяющим использовать все возможности технологий. С помощью своих продуктов и услуг мы стараемся снять покров таинственности с искусственного интеллекта и технологий и сделать их доступными, понятными и пригодными для использования всеми желающими.

Наш опыт

У нас богатый опыт использования технологий. С одной стороны, мы умеем создавать SaaS, такие как CuantumAI, и благодаря своим обширным знаниям и навыкам в области веб-разработки стараемся предлагать передовые и интуитивно понятные приложения. Мы стремимся использовать потенциал искусственного интеллекта для решения практических задач и повышения эффективности бизнеса.

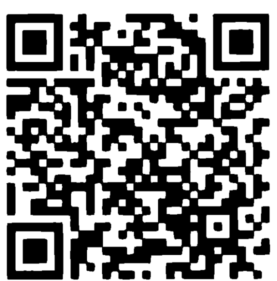
С другой стороны, мы — преподаватели, преданные своему делу. Наши книги дают глубокое представление о различных языках программирования и искусственном интеллекте и помогают новичкам и опытным программистам расширить свои знания и навыки. Мы гордимся своим умением эффективно распространять знания, излагая сложные концепции понятным языком.

Более того, наше мастерство в создании интерактивных веб-интерфейсов не имеет себе равных. Используя сочетание HTML, CSS, JavaScript и Three.js, мы создаем захватывающую и привлекательную цифровую среду, которая очаровывает пользователей и выводит их опыт работы онлайн на новый уровень.

Сотрудничая с Cuantum Technologies, вы не просто получаете услугу или продукт — вы вступаете вместе с нами в будущее, где каждый желающий сможет улучшать свою жизнь с помощью технологий и искусственного интеллекта.

ИСХОДНЫЙ КОД ПРИМЕРОВ

Чтобы упростить процесс обучения, мы разместили в Интернете все примеры программного кода, приведенные в этой книге. Перейдя по ссылке, представленной ниже, вы сможете получить доступ к обширной базе использованного кода. Это позволит вам не только копировать код, но и просматривать и анализировать его в удобное для вас время. Надеемся, что благодаря этому дополнительному ресурсу вы лучше поймете описанные в книге идеи и сможете беспрепятственно обучаться.



https://books.cuantum.tech/introduction-algorithms/code/

ПРЕДИСЛОВИЕ

Дорогие читатели, вам предстоит увлекательное путешествие по миру алгоритмов! Цель данной книги — раскрыть его тайны и осветить пути, ведущие к решению многих задач и в итоге к более эффективному программированию. Каким бы ни был уровень ваших знаний или опыта, благодаря этой книге вы сможете понимать и оценивать алгоритмы и, как результат, освоите их.

Алгоритмы — это, по сути, процедуры решения задач и незаменимые инструменты для навигации по обширным пространствам данных и информации. Алгоритмы подобны рецептам; они дают пошаговые инструкции для решения задач или достижения целей. Язык этих инструкций, в отличие от кулинарного рецепта, основан на математике и логике, которые компьютеры хорошо понимают.

Наше путешествие мы начнем со знакомства с понятием алгоритмов, сделав основной упор на его четком и кратком определении. Затем рассмотрим разные типы алгоритмов: поисковые, помогающие находить определенные данные в наборе; сортировочные, позволяющие организовать данные определенным образом; а также алгоритмы построения графов, которые помогают понять и решить задачи, связанные с сетями и отношениями.

После этого мы углубимся в различные методы, используемые при разработке алгоритмов, такие как рекурсия, когда алгоритм вызывает себя для решения меньших экземпляров одной и той же задачи, и итерации, когда задачи решаются с помощью циклов. Вдобавок мы познакомимся с обратной трассировкой, которая подразумевает выполнение ряда решений и отмену того или иного выбора, когда оказывается, что он неэффективен. Кроме того, мы уделим внимание методу ветвей и границ, который используется в задачах оптимизации.

Чтобы закрепить все это, мы рассмотрим структуры данных, которые используются в алгоритмах: массивы, связанные списки, стеки, очереди, деревья и графы. Каждая из структур обладает уникальным набором характеристик, который делает ее наиболее подходящей для определенных задач и приложений.

Разобравшись с особенностями конструирования и функционирования алгоритмов, мы перейдем к их применению в реальных условиях, изучим их использование в базах данных, искусственном интеллекте и сетевой маршрутизации. Мы убедимся в том, что многие технологии, которые сегодня считаются обыденными, эффективно функционируют именно благодаря возможностям алгоритмов.

Закрепить полученные знания, применить их на практике и развить способности к решению задач вам помогут примеры, размещенные в конце каждой главы. Ведь изучать алгоритмы означает не только запоминать теорию, но и обретать практические навыки, позволяющие эффективно решать реальные задачи.

Обратите внимание: эта книга содержательная и довольно подробная, однако написана простым и доступным языком. Ее цель не в том, чтобы напичкать вас информацией, а в том, чтобы вооружить вас знаниями и показать, что решение даже довольно сложных задач может быть посильным и увлекательным занятием. Кем бы вы ни были — студентом, желающим лучше понять алгоритмы, профессионалом, стремящимся углубить свои знания, или просто человеком, который интересуется математическими и логическими механизмами, управляющими нашим цифровым миром, — эта книга для вас.

Наше путешествие будет трудным и вместе с тем очень полезным. Цифровой мир и используемые в нем методы решения задач откроются вам с новой стороны. Вы научитесь обращать внимание на эффективность алгоритмов и писать более действенный код. Но самое главное — вы сможете стать более уверенным и компетентным специалистом по решению задач.

Итак, дорогие читатели, открывая книгу и отправляясь в путешествие, вспомните слова великого Эдсгера Дейкстры: «Алгоритмическое мышление — это квинтэссенция программирования». Мы надеемся, что к концу книги вы оцените истинность этого утверждения и захотите продолжить обучаться и развивать свое алгоритмическое мышление.

Добро пожаловать в интригующий мир алгоритмов! Начнем это приключение вместе. Приятного обучения!

ВВЕДЕНИЕ

О чем эта книга

Если вы взяли в руки эту книгу, то, скорее всего, вы — студент, преподаватель, профессионал или любитель, интересующийся сложным, но увлекательным миром алгоритмов. Цель книги — познакомить вас с основами алгоритмов и помочь оценить их возможности и элегантность.

Алгоритм, по сути, представляет собой набор инструкций для решения задачи. Он во многом похож на рецепт: это пошаговое руководство, неукоснительное следование которому гарантирует решение. Но, как и в кулинарии, создание алгоритмов — это искусство. Одну и ту же задачу можно решить множеством способов, но одни из них более эффективны, элегантны или просты, чем другие.

Цель этой книги — дать знания и навыки, которые позволят вам понимать, анализировать и писать эффективные и полезные алгоритмы. Создавая ее, мы стремились к тому, чтобы материал был понятным для читателей — новичков в информатике, а также служил подспорьем для тех, кто уже имеет опыт в этой области.

Книга структурирована так, чтобы ваше понимание темы постепенно расширялось. Мы начнем с основ: определим, что такое алгоритмы, и проиллюстрируем их важность в информатике. Затем углубимся в искусство выражения алгоритмов с помощью псевдокода и блок-схем.

По мере изучения глав вы познакомитесь с разными алгоритмами: поиска и сортировки, построения графов и т. д. Вы изучите их внутреннее устройство, узнаете, как выбрать лучший алгоритм для конкретной задачи, и увидите, как эти алгоритмические методы используются в реальных приложениях.

Более того, эта книга не только познакомит вас с теорией. Мы считаем, что лучший способ освоить алгоритмы — применять их на практике. Поэтому каждая глава завершается практическими задачами. Чтобы решить их, вам понадобится использовать полученные знания. Задачи сопровождаются

готовыми решениями, чтобы вы могли проверить себя и убедиться в том, что правильно понимаете материал.

К концу этой книги вы освоите основные алгоритмы, что позволит вам эффективно использовать их для решения сложных задач в ваших учебных, профессиональных или личных проектах.

Независимо от того, стремитесь ли вы стать разработчиком программного обеспечения, аналитиком данных или просто любите решать головоломки, — эта книга даст вам все необходимые инструменты. Надеемся, что она окажется для вас поучительной, увлекательной и вдохновляющей, когда вы начнете или продолжите свое путешествие по удивительному миру информатики.

Кому адресована книга

Эта книга предназначена для максимально широкого круга людей, которым интересен мир информатики и алгоритмов. Мы писали ее для следующих категорий читателей.

- **Учащиеся.** Если вы школьник, только начинающий изучать информатику, или студент высшего учебного заведения — эта книга послужит вам учебником, в котором в простой и доступной форме объясняются фундаментальные алгоритмические идеи и методы. Речь идет о темах, составляющих основу многих учебных программ по информатике, принятых во всем мире.
- **Преподаватели.** Если вы преподаватель, обучающий новые поколения ученых-информатиков, то эта книга станет для вас ценным источником сведений. Благодаря пошаговым объяснениям, примерам из реальной жизни и практическим задачам она послужит отличным справочником по планированию учебной программы и полезным дополнением к учебным лекциям.
- **Профессионалы.** Вы разработчик программного обеспечения, аналитик данных или профессионал в области технологий, желающий улучшить свое алгоритмическое мышление и навыки решения задач? Или, может быть, вы готовитесь к техническому собеседованию, посвященному структурам данных и алгоритмам? Эта книга послужит источником необходимых знаний и поможет освежить в памяти фундаментальные понятия.
- **Самоучки.** Если вы любитель, обучающийся самостоятельно (возможно, вы подумываете о смене карьеры или являетесь программистом-самоучкой,

который хочет лучше понимать алгоритмы), то книга поможет вам в этом и гарантирует, что вы получите базовые знания.

- **Предприниматели в сфере технологий.** Основатели стартапов и менеджеры по продукту, работающие в технологических компаниях, благодаря базовому пониманию алгоритмов смогут принимать более взвешенные решения о разработке продукта, видеть возможности и ограничения своего программного обеспечения и более эффективно взаимодействовать с техническими специалистами в команде.

Проще говоря, книга предназначена для всех, кто хочет понять, что такое алгоритмы. Мы считаем, что их может изучить каждый, и приложили все усилия, чтобы сделать материал максимально доступным и интересным.

Если вы не уверены в своих математических познаниях или являетесь новичком в программировании — не волнуйтесь. Мы начнем с самых основ и простым и понятным языком объясним все необходимые математические или программные концепции. Хорошо, если у вас есть опыт программирования, но это не обязательное условие. Основное внимание мы уделим описанию концепций, а для иллюстрации идей используем псевдокод — простые обобщенные представления алгоритмов.

Читая эту книгу, помните: изучение алгоритмов представляет собой не просто запоминание процедур, а освоение нового способа мышления и решения задач. Решайте задачи, наслаждайтесь процессом и не бойтесь совершать ошибки. Именно так мы учимся, совершенствуемся и в результате осваиваем любые области.

Мы рады, что вы отправились с нами в это приключение по миру алгоритмов, и нам не терпится увидеть, куда вас приведут новые знания!

Как пользоваться изданием

Нам хотелось бы, чтобы эта книга была вам максимально полезной независимо от вашего стиля и темпа обучения. Вот несколько советов о том, как работать с ней эффективно.

- **Читайте последовательно.** Главы выстроены в таком порядке, что каждая последующая основана на понятиях, представленных в предыдущих главах. Если вы новичок в теме изучения алгоритмов, то мы настоятельно рекомендуем читать книгу с самого начала и последовательно. Это позволит глубоко понять каждую тему.

- **Учитесь активно.** Мы верим в принцип, согласно которому лучший способ учиться — это действовать. В конце каждой главы содержится множество практических задач и упражнений. Не пропускайте их! Они помогут вам применить новые навыки на практике, понять нюансы и выявить пробелы в знаниях. Помните: верный ответ может прийти не сразу; часто больше всего знаний вы можете получить благодаря именно тем задачам, которые сначала решили неправильно. Поработайте над задачами, найдите свое решение, а затем сравните его с приведенным.
- **Делайте заметки и подводите итоги главы.** Мы рекомендуем в процессе чтения глав делать заметки, записывать вопросы, выделять важные моменты и своими словами излагать резюме того, что вы узнали. Это поможет вам лучше понять материал и облегчит его дальнейшее изучение.
- **Выделяйте время для чтения.** Как и любой новый предмет, изучение алгоритмов требует времени и сосредоточенности. Выделите в своем расписании время, которое посвятите чтению и практике. Даже если вы будете заниматься всего несколько часов в неделю, это может существенно улучшить ваши навыки.
- **Делайте перерывы и размышляйте.** Не спешите. Делайте перерывы, размышляйте над тем, что узнали, и позвольте новым идеям усвоиться. Иногда самые сложные идеи становятся понятными после того, как мы отдохнули или прогулялись в парке.
- **Обсуждайте и делитесь мнениями.** Если это возможно, то обсуждайте идеи и задачи с коллегами, товарищами или одноклассниками. Объяснять материал другим людям — отличный способ понять его еще лучше, к тому же в процессе обсуждения вы можете посмотреть на задачу с разных точек зрения и получить новые идеи.
- **Экспериментируйте.** Алгоритмы — это не просто теоретические конструкции, они предназначены для использования в реальных приложениях. Экспериментируйте с реализацией алгоритмов на любом языке программирования по вашему выбору. Посмотрите, как изменение входных данных влияет на поведение и производительность алгоритма.

Наша книга — инструмент, и то, как вы им воспользуетесь, зависит лишь от вас. Будьте терпимы к себе и помните: обучение — это не гонка, а приятное путешествие. Не все будут двигаться в одинаковом темпе, и это нормально. Самое главное — воспитать в себе интерес и любовь к учебе.

Досконально изучите каждую концепцию, прежде чем двигаться дальше. Алгоритмы подобны головоломкам, и их решение может приносить огромное

удовольствие. Сохраняйте позитивный настрой и помните: каждый великий программист, специалист по данным или ученый-информатик когда-то тоже не знал, что такое алгоритм. Будьте настойчивыми и практикуйтесь — и также сможете овладеть искусством алгоритмов.

Приятного путешествия по увлекательному миру алгоритмов!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1
ВВЕДЕНИЕ В АЛГОРИТМЫ

https://t.me/it_boooks/2

В этой главе мы рассмотрим базовые понятия, которые послужат основой для остальной части книги.

Для начала определим, что такое алгоритм и почему так важно это понимать. Алгоритмы — это, по сути, наборы инструкций, с помощью которых компьютер или человек может решать задачи. Алгоритмы являются строительными блоками современных технологий и используются повсюду: от поисковых систем до беспилотных автомобилей.

Затем мы рассмотрим некоторые характеристики хороших алгоритмов. К ним относятся, например, эффективность, точность и адаптируемость. Поняв, что это такое, вы будете готовы приступить к созданию собственных эффективных алгоритмов.

На протяжении всей главы мы будем приводить примеры и давать упражнения, которые помогут вам усвоить фундаментальные понятия. К концу главы вы получите четкое представление о них и будете готовы приступить к изучению более сложных тем.

1.1. Что такое алгоритм

Алгоритм — это набор инструкций, пошаговая процедура, которую можно применять к различным видам задач. Задачами может быть что угодно: от выпечки торта до нахождения наибольшего общего делителя двух чисел.

Алгоритмы могут быть простыми или очень сложными в зависимости от решаемой задачи. Их можно использовать в самых разных областях — от математики до информатики — для достижения конкретных целей. Фактически алгоритмы являются важной частью программирования и применяются при создании программ, работающих на компьютерах и других электронных устройствах. Помимо этого, алгоритмы используются и в таких областях, как инженерия, медицина и финансы, для решения сложных задач.

Прежде чем погружаться в вычислительные аспекты, рассмотрим стандартный пример.

Пример

Предположим, к вам в гости должен прийти друг и вы решили испечь шоколадный кекс. Простой алгоритм, которому можно следовать, выглядит так.

1. Подготовьте все необходимые ингредиенты (мука, яйца, сахар, какао-порошок, разрыхлитель и т. д.).
2. Разогрейте духовку до 175 градусов.
3. В одной миске смешайте сухие ингредиенты.
4. В другой миске взбейте яйца, а затем добавьте их в миску с сухими ингредиентами и перемешайте.
5. Вылейте тесто в смазанную маслом форму для кекса.
6. Выпекайте 30 минут.
7. Проверьте готовность кекса, проткнув его зубочисткой: если она чистая, значит, кекс готов.
8. Перед подачей дайте кексу остыть.

Этот список представляет собой простой алгоритм выпечки кекса, пошаговую процедуру, при правильном выполнении которой должен получиться восхитительный шоколадный кекс.

В информатике алгоритм решает вычислительные задачи. Он принимает входные данные, обрабатывает их, выполняя ряд вычислительных шагов, и выдает результат.

Рассмотрим вычислительный пример.

Пример

Возьмем задачу поиска наибольшего числа в списке. Простой алгоритм, решающий ее, выглядит так.

1. Считать первое число в списке самым большим.
2. Для каждого оставшегося числа в списке: если это число больше текущего самого большого числа, то считать его новым самым большим числом.
3. После проверки всех чисел текущим наибольшим числом будет самое большое число в списке.

Рассмотрим список чисел: 5, 3, 9, 1, 7. Этот алгоритм будет работать следующим образом.

1. Сначала мы предполагаем, что число 5 (первое) является самым большим.
2. Затем сравниваем 5 и 3 (следующее число). Число 5 больше, поэтому остается текущим наибольшим числом.
3. Далее сравниваем 5 и 9. Число 9 больше, поэтому становится новым текущим наибольшим числом.
4. Затем сравниваем 9 с 1 и 7. Поскольку 9 больше обоих чисел, то остается текущим наибольшим числом.
5. Проверив все числа, мы приходим к выводу, что 9 — самое большое число в списке.

Пример очень простой, но иллюстрирует основную идею: алгоритм — это пошаговая процедура решения задачи.

1.1.1. Характеристики хорошего алгоритма

С точки зрения информатики хороший алгоритм — не просто процедура, которая позволяет правильно решить задачу. Это тщательно продуманная последовательность шагов, благодаря которой можно не только получить правильное решение, но и достичь масштабируемости, эффективности и гибкости.

Хороший алгоритм должен уметь обрабатывать большие наборы данных и приспосабливаться к меняющимся обстоятельствам или требованиям без необходимости полной переделки. Более того, хороший алгоритм должен быть понятным и простым в сопровождении, иметь точную документацию и хорошо организованный код, позволяющий легко изменять и отлаживать его.

Проще говоря, хороший алгоритм не просто решает одну задачу, а служит надежной и гибкой основой, позволяющей решать множество задач в различных контекстах.

Алгоритм также должен иметь следующие характеристики.

- **Однозначность.** Любой набор инструкций должен быть однозначным. То есть каждый шаг должен быть понятным и иметь только одно

возможное толкование. Неоднозначность приводит к путанице и ошибкам, которые могут помешать достижению желаемого результата.

Чтобы инструкции были однозначными, важно использовать четкие и выразительные формулировки, избегать слишком сложных слов и фраз и тщательно продумывать порядок и структуру шагов. Кроме того, полезно попросить кого-нибудь со стороны просмотреть инструкции, чтобы убедиться, что они понятны и им легко следовать.

- **Детерминированность.** Детерминированные алгоритмы предназначены для получения одного и того же результата для одних и тех же входных данных и выполнения одних и тех же шагов в одном и том же порядке. Это означает, что если вы дважды введете одни и те же данные, то оба раза получите один и тот же результат.

  Это свойство чрезвычайно значимо во многих областях исследований, таких как информатика, где важно иметь возможность воспроизводить результаты и гарантировать получение предсказуемых выходных данных. При выполнении последовательности заранее определенных шагов детерминированные алгоритмы могут обеспечить такой уровень предсказуемости и контроля, который невозможен при использовании недетерминированных алгоритмов.

  Поэтому крайне важно при разработке алгоритма гарантировать его детерминированность, чтобы его результатам можно было доверять.

- **Конечность.** Конечные алгоритмы предназначены для решения конкретной задачи при заданных входных данных и всегда должны завершаться после конечного количества шагов. Это требование важно, поскольку гарантирует, что алгоритм не будет работать бесконечно, потребляя вычислительные ресурсы и вызывая проблемы с производительностью.

  Чтобы удовлетворить требованию конечности, алгоритм следует разрабатывать с использованием набора правил и процедур, гарантирующих его завершение за конечное количество шагов. Обычно это количество зависит от объема входных данных и сложности алгоритма. Поэтому важно его оптимизировать, чтобы свести количество шагов к минимуму и при этом получить желаемый результат.

- **Выполнимость.** Чтобы проект был успешным, важно гарантировать, что он не только хорошо спланирован, но и осуществим. То есть он должен быть простым настолько, чтобы его можно было выполнить, используя имеющиеся ресурсы.

Кроме того, следует учитывать любые потенциальные проблемы или ограничения, которые могут возникнуть на этапе реализации, и иметь план конструктивных действий на случай непредвиденных обстоятельств.

Тщательный подход к планированию и реализации позволяет сделать проект более осуществимым и в конечном счете более успешным.

- **Независимость.** Для ее достижения важно, чтобы алгоритм содержал пошаговые инструкции, не предполагающие применения конкретного программного кода.

  Это позволит широкому кругу пользователей (независимо от их уровня подготовки) понимать алгоритм и реализовывать его. Более того, алгоритм, независимый от программного кода, будет легче изменять и обновлять в будущем, что позволит сделать его более гибким и адаптируемым к меняющимся потребностям и обстоятельствам.

Вернемся к примеру поиска наибольшего числа в списке. Наш алгоритм отвечает всем пяти критериям: он однозначный и детерминированный, завершает работу после конечного количества шагов (после проверки всех чисел), выполним (найти наибольшее число в списке — простая задача, которую может выполнить любой компьютер) и не зависит от конкретного языка программирования.

Есть еще две очень важные характеристики хороших алгоритмов — **эффективность** и **масштабируемость**. Мы обсудим их более подробно в следующих главах, а пока просто запомните: помимо вышесказанного, хороший алгоритм должен эффективно решать свою задачу (используя как можно меньше вычислительных ресурсов) и хорошо масштабироваться (продолжать эффективно работать даже при увеличении объема входных данных).

Понимание этих характеристик и создание алгоритмов, обладающих ими, — ключевая часть изучения информатики, и именно на этом мы будем концентрироваться на протяжении всей книги. В следующем подразделе мы рассмотрим, почему алгоритмы так важны и как они используются в вычислениях.

1.1.2. Как используются алгоритмы

Алгоритмы играют важную роль в информатике и многих других областях. Они используются в приложениях самой разной сложности, и их значение невозможно переоценить. Алгоритмы служат для решения проблем, автоматизации задач и прогнозирования.

Кроме того, их используют для анализа данных, оптимизации процессов и создания искусственного интеллекта. Технологии все больше влияют на нашу повседневную жизнь. Параллельно возрастает и значимость алгоритмов. В будущем можно ожидать, что они продолжат играть крайне важную роль в развитии многих областей, в том числе медицины, финансов и инженерии. Поэтому нужно иметь четкое представление об алгоритмах и их применении. Это позволит добиться успеха в постоянно развивающемся мире технологий.

Как вы уже знаете, с точки зрения информатики хороший алгоритм — не просто процедура, правильно решающая задачу. Помимо этого, чтобы быть полезным, он должен обладать множеством других характеристик, особенно в областях, перечисленных ниже.

- **Поисковые системы.** Когда вы вводите запрос в Google, Bing или другую поисковую систему, их алгоритмы начинают работать, чтобы определить, какие веб-страницы наиболее точно соответствуют запросу. Эти алгоритмы сложны и учитывают множество факторов, таких как ключевые слова на страницах, релевантность этих слов вашему запросу, качество контента и популярность страницы.

  В ходе данного процесса сканируются и индексируются миллиарды веб-страниц, поэтому поисковые системы используют большие вычислительные мощности и объемы ресурсов, чтобы вы могли получить наиболее релевантные результаты.

- **Социальные сети.** Платформы социальных сетей используют сложные алгоритмы, чтобы определить, какой контент отображать в вашей ленте новостей. Эти алгоритмы учитывают несколько факторов, например, тип контента, с которым вы обычно взаимодействуете, время суток, когда вы наиболее активны, и контент, который вы опубликовали недавно.

  Кроме того, эти алгоритмы постоянно подвергают обновлениям и улучшениям, чтобы гарантировать актуальность доставляемого вам контента. В результате социальные сети стали неотъемлемой частью нашей повседневной жизни, позволяя оставаться на связи с друзьями и семьей, находить новую информацию и взаимодействовать с нашими любимыми брендами и некими людьми, используя индивидуальный подход.

- **Электронная торговля.** В эпоху цифровизации онлайн-покупки стали весьма популярны среди потребителей. Amazon, один из ведущих интернет-магазинов, использует алгоритмы рекомендаций, чтобы предлагать

товары клиентам. Эти алгоритмы анализируют историю посещений и покупки клиента и генерируют предложения на основе его интересов и предпочтений.

Используя эти алгоритмы, Amazon смог организовать обслуживание на основе индивидуального подхода к клиенту, что привело к повышению удовлетворенности и лояльности покупателей. Чтобы повысить продажи и улучшить качество обслуживания клиентов, другие интернет-магазины тоже начали использовать эти алгоритмы, что сделало их важнейшим аспектом индустрии электронной торговли.

- **GPS.** GPS (Global Positioning System — система глобального позиционирования) — спутниковая навигационная система, которая предоставляет информацию о местоположении и времени при любых погодных условиях, в любой точке Земли или вблизи нее. GPS использует алгоритмы для анализа данных, собранных с нескольких спутников, чтобы определить самый короткий или самый быстрый маршрут до места назначения.

  Технология GPS произвела революцию в области навигации и путешествий, упрощая и повышая эффективность перемещения из одного места в другое. Система используется не только в телефонах и автомобильных навигаторах, но и в различных отраслях промышленности, таких как авиация, сельское хозяйство и транспорт, для отслеживания техники, оборудования и персонала и управления ими.

  Эта технология продолжает развиваться, поэтому ее возможности по улучшению нашей повседневной жизни будут только расти.

- **Машинное обучение.** Это важнейший аспект искусственного интеллекта. В данном виде обучения используются алгоритмы, которые учатся на основе опыта и данных, что позволяет им решать такие сложные задачи, как распознавание голоса, классификация изображений и генерирование рекомендаций. Роль машинного обучения в современную цифровую эпоху неоспорима, и оно применяется в широком спектре приложений: от беспилотных автомобилей до персонализированного маркетинга.

  Благодаря тому, что в машинном обучении используются большие объемы данных и сложные алгоритмы, изменяются различные отрасли, наша жизнь, работа и способы взаимодействия с технологиями. И если вы интересуетесь будущим технологий и хотите быть на шаг впереди, то просто обязаны осваивать машинное обучение.

- **Медицина.** В этой области алгоритмы также играют все более важную роль. Они не только позволяют улучшать медицинские снимки и находить аномалии, но и помогают врачам ставить более точные диагнозы.

  Эти алгоритмы проводят анализ больших объемов медицинских снимков и записей в историях болезней пациентов, благодаря чему могут помочь врачам предсказать возможность появления заболевания еще до его возникновения, позволяя проводить более раннее вмешательство и повышать шансы на выздоровление.

  Алгоритмы используются для разработки индивидуальных планов лечения пациентов с учетом их уникальной истории болезни, генетического профиля и образа жизни. Медицина продолжает развиваться, поэтому можно ожидать, что алгоритмы будут играть еще более важную роль в лечении пациентов и расширении медицинских знаний.

В этих и многих других областях применения эффективность и точность алгоритма могут существенно влиять на качество результатов. Плохой алгоритм, в отличие от хорошего, может работать слишком медленно и выдавать неточные, а то и бесполезные результаты. Вот почему важно разбираться в алгоритмах и уметь их разрабатывать, а также анализировать их эффективность.

1.1.3. Краткое подведение итогов и некоторые моменты для размышления

Напомним, что алгоритм представляет собой пошаговую процедуру решения задачи. Это не просто набор инструкций: алгоритм должен быть однозначным, детерминированным, конечным, выполнимым и независимым. Алгоритмы являются фундаментальным понятием информатики и играют решающую роль в различных аспектах нашей жизни.

Например, поисковые системы используют алгоритмы, чтобы вернуть релевантные результаты в ответ на наши поисковые запросы. Платформы социальных сетей задействуют алгоритмы, чтобы показывать контент, который, по их мнению, нас заинтересует. Системы GPS используют алгоритмы, чтобы определить оптимальный маршрут до места назначения. По мере роста цифровизации умение разбираться в алгоритмах становится все более важным.

Изучая алгоритмы, можно понять, как они работают и как с их помощью упростить нашу жизнь и сделать ее более эффективной. Эти знания ценны и широко применимы, поскольку алгоритмы используются во множестве областей: финансовой, медицинской, транспортной и др.

По мере изучения информации, возможно, будет полезно задуматься над следующими вопросами.

1. Можно ли типичную повседневную задачу или некое рутинное действие определить как алгоритм? Если бы вам пришлось разложить данный процесс на шаги, то как бы вы это сделали? Есть ли какие-то шаги, которые можно разбить на еще более мелкие или упростить?
2. Подумайте о качествах, которые делают алгоритм эффективным. Бывали ли у вас случаи, когда более строгое соблюдение этих принципов могло привести к более действенному процессу или решению задачи? Подумайте, как внедрение этих качеств в вашу собственную работу может улучшить будущие результаты.
3. Учитывая широту и разнообразие применения исследованных нами алгоритмов, стоит задуматься о том, как более глубокое понимание этих систем может изменить ваш подход к взаимодействию с ними. По вашему мнению, каким образом вы могли бы изменить взаимодействие с этими системами в свете такого понимания?

В следующих разделах мы углубимся в нюансы различных алгоритмов. Начав с более простых алгоритмов, постепенно перейдем к более сложным. Это позволит вам получить полное представление о теме. Кроме того, вы будете учиться анализировать эффективность алгоритмов, что является важным навыком в информатике и алгоритмическом мышлении.

Однако прежде чем это произойдет, важно задуматься об основах алгоритмов. Они являются строительными блоками информатики и программирования, и их нужно понимать, чтобы быть успешным в этих областях. Уделите некоторое время размышлениям о значении алгоритмов и о том, почему они так важны.

Помните: обучение — это путешествие, и вы делаете первый шаг. Впереди вас ждет интересный путь, полный открытий и испытаний. Однако, следуя этому пути, без колебаний возвращайтесь к любому разделу книги, если почувствуете такую необходимость. Это поможет вам закрепить знания. Удачи и приятного обучения!

1.2. Важность алгоритмов в информатике

Теперь, получив некоторое представление о том, что такое алгоритмы, важно осознать их значимость в информатике и понять причины, по которым их изучение является важнейшим компонентом любой учебной программы по информатике.

Информатика — интересная наука, занимающаяся вопросами решения задач. В каждой программе, компьютерной игре или цифровом интерфейсе скрыто работают алгоритмы, благодаря которым вы достигаете желаемых результатов.

Изучать алгоритмы важно не только потому, что это помогает понять работу компьютеров, — вдобавок вы учитесь разрабатывать и анализировать алгоритмы, которые могут эффективно решать сложные задачи. Как вы уже знаете, алгоритмы находят применение в различных областях, таких как анализ данных, машинное обучение и искусственный интеллект, что делает их ценным инструментом, которым пользуются исследователи и ученые во всем мире.

Поэтому крайне важно иметь полное представление об алгоритмах и их роли в информатике, чтобы добиться успеха в этой области. Знание алгоритмов поможет получить представление о работе компьютеров и о том, как с их помощью эффективно решать реальные задачи.

1.2.1. Алгоритмы лежат в основе нашего цифрового мира

Компьютеры — сложные машины, работающие с высочайшей скоростью и выполняющие миллиарды операций в секунду. Однако полезность компьютеров зависит от алгоритмов, которые ими управляют. Без алгоритмов компьютеры подобны мощному автомобилю без водителя.

Алгоритмы нужны компьютерам, поскольку организуют данные в структуры и дают инструкции по их обработке. Например, когда вы выполняете поиск в Google, алгоритм сортирует миллиарды веб-страниц и выдает результаты за считаные секунды. Аналогично, когда вы используете GPS-навигацию, алгоритм вычисляет кратчайший или самый быстрый путь к месту назначения. А когда вы смотрите фильм на Netflix, алгоритм буферизует видео и регулирует качество в зависимости от скорости вашего интернет-соединения.

Важно отметить, что использование алгоритмов не ограничивается только этими задачами. Они применяются во многих других приложениях, что делает их неотъемлемой частью компьютерного мира. Таким образом, алгоритмы являются важнейшим компонентом, обеспечивающим бесперебойную и эффективную работу компьютеров.

1.2.2. Алгоритмы повышают эффективность

Еще одна ключевая причина, почему алгоритмы так важны, — они оказывают существенное влияние на эффективность. Когда дело доходит до решения задач, разница между хорошим и плохим алгоритмом может быть астрономической, особенно по мере увеличения сложности задачи.

Например, рассмотрим сценарий, когда компания пытается отсортировать большой объем данных, чтобы выявить значимую информацию и извлечь ее. С одной стороны, решение этой задачи с помощью неэффективного алгоритма может потребовать чересчур большого количества времени и ресурсов, что приведет к задержкам и потенциально упущенным возможностям.

С другой стороны, высокоэффективный алгоритм позволяет решить ту же задачу за существенно меньшее время, принимать более быстрые решения на основе данных и опережать конкурентов. Поэтому для решения задач крайне важно использовать подходящие алгоритмы и гарантировать их максимальную эффективность.

Пример

Допустим, вы написали программу, которая находит максимальное число в списке. Одним из потенциальных алгоритмов может быть сортировка списка в порядке возрастания, а затем выбор последнего числа (поскольку оно будет самым большим). Этот алгоритм действительно решит задачу, но не будет самым эффективным. Сортировка списка требует большого количества вычислительного времени, особенно если список большой.

Как уже говорилось выше, эффективнее пройти по списку только один раз, запоминая максимальное число, встретившееся на данный момент. Такой алгоритм достигает цели, но работает гораздо быстрее и использует меньше вычислительных ресурсов.

Приведенный выше пример демонстрирует, почему важно не просто решать задачу, а делать это эффективно. Если на решение задачи уходит несколько

часов и при этом полностью исчерпываются ресурсы памяти — такой подход неэффективен. И наоборот, решать задачу за долю секунды, затрачивая разумный объем памяти, — очень эффективно.

```
def find_max(numbers):
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num

numbers = [5, 3, 9, 1, 7]
print (find_max(numbers)) # Выведет: 9
```

В этом фрагменте кода на Python мы реализуем эффективный алгоритм поиска максимального числа в списке. Даже при работе с большим списком чисел этот алгоритм будет работать намного быстрее, чем решение на основе сортировки.

1.2.3. Алгоритмы лежат в основе передовых технологий

Как вы уже знаете, алгоритмы являются фундаментальными строительными блоками информатики и используются во многих передовых областях, таких как искусственный интеллект, машинное обучение и наука о данных. Значение алгоритмов в области информатики невозможно переоценить.

Например, алгоритмы машинного обучения предназначены для изучения закономерностей в данных и составления прогнозов или принятия решений, при которых выполнение задачи не требует явного программирования. Это сложный процесс, требующий разработки и реализации мощных алгоритмов. В науке о данных алгоритмы служат для анализа больших наборов данных и извлечения информации, которая затем используется для принятия обоснованных решений.

Но алгоритмы играют очень важную роль не только в науке о данных и машинном обучении. В криптографии они используются для защиты коммуникаций и транзакций, а в компьютерной графике — для создания потрясающих визуальных эффектов. В сетевой маршрутизации алгоритмы позволяют определить оптимальные пути передачи данных по сети, а в вычислительной биологии — помогают понять сложность биологических систем.

Алгоритмы имеют очень широкое применение, и от них зависит структурирование задач и их эффективное решение. Таким образом, очевидно, что везде, где есть задача, требующая решения, возникает вероятность того, что в этом процессе алгоритмы будут играть крайне важную роль.

1.2.4. Будущие последствия и достижения в сфере алгоритмов

Алгоритмы составляют неотъемлемую часть современных вычислительных систем, и по мере появления новых технологий их значимость будет только возрастать. Например, квантовые вычисления позволяют предположить, что скорость выполнения определенных задач будет расти экспоненциально, но их развитие опирается на создание новых алгоритмов, которые могут эффективно работать с уникальными свойствами квантовых систем. Кроме того, такие области, как искусственный интеллект и машинное обучение, продолжают быстро развиваться, отчасти благодаря все более сложным алгоритмам.

Несмотря на многочисленные преимущества, сложные алгоритмы порождают и новые проблемы, в том числе этические. Так, алгоритмы машинного обучения могут непреднамеренно обучаться и отдавать предпочтение необъективным сведениям, присутствующим в обучающих данных, что приводит к выдаче искаженных результатов. Алгоритмы, используемые при принятии важных решений, таких как одобрение кредита или вынесение приговора к тюремному заключению, должны быть простыми и понятными, но это можно гарантировать не всегда.

Поэтому изучение алгоритмов подразумевает не только умение эффективно решать задачи, но и понимание последствий наших решений, осознание потенциальных ловушек и предубеждений и разработку объективных и понятных систем. Чтобы достичь этой цели, необходимо постоянно совершенствовать алгоритмы, используемые в различных областях, обеспечивать их эффективность, объективность и понятность. Кроме того, важно информировать людей о роли алгоритмов в нашей жизни и их влиянии на общество, чтобы решения об их разработке и использовании принимались обоснованно.

По мере погружения в мир алгоритмов поразмышляйте над следующими вопросами.

1. Как все более широкое использование алгоритмов в различных аспектах нашей жизни может повлиять на наше общество? Какие потенциальные преимущества и недостатки вы видите?

2. Какую роль вы как информатик или человек, изучающий алгоритмы, играете в решении этических проблем, связанных с алгоритмами?
3. Как можно было бы гарантировать, что разрабатываемые вами алгоритмы будут не только эффективными и полезными, но и объективными и понятными?

Углубляясь в материал этой книги и знакомясь с различными алгоритмами, мы обсудим не только технические аспекты их эффективности и варианты использования, но и другие нюансы. Мы подробно рассмотрим, как алгоритмы меняют цифровой мир и влияют на нашу повседневную жизнь, помогают принимать правильные решения, повышают нашу продуктивность и даже меняют наше социальное взаимодействие.

Важно понимать, что освоение алгоритмов — навык не только технический, но и когнитивный, требующий критического мышления и творческого подхода. Алгоритмы продолжают проникать в нашу жизнь, поэтому для нас все более важным становится умение правильно оценивать их влияние на мир. Начиная с поисковых систем и заканчивая платформами социальных сетей — алгоритмы стали основой нашего цифрового ландшафта.

На этом мы завершаем раздел о значении алгоритмов в информатике. Теперь, осознав их важность, вы лучше понимаете, почему мы их изучаем и по какой причине они играют такую большую роль в этой области.

Как уже было сказано ранее, изучение алгоритмов — важнейший аспект информатики. Они позволяют выдавать нашим машинам инструкции о том, как эффективно выполнять задачи, что является основой работы в этой области. Алгоритмы используются в различных сферах жизни, таких как финансы, здравоохранение, транспорт и многие другие.

Освоив алгоритмы, вы сможете писать более качественный код, эффективнее решать задачи и лучше понимать, как устроен цифровой мир. Более того, изучение алгоритмов поможет вам развить навыки критического мышления и решения задач, которые пригодятся не только в области информатики, но и в других сферах жизни.

Кроме того, глубокое понимание алгоритмов может дать вам преимущество на рынке труда, поскольку это очень востребованный навык. Поэтому время, потраченное на их изучение, окупится сторицей и принесет вам много пользы как в личной, так и в профессиональной жизни.

1.3. Основы вычислительного мышления

Вычислительное мышление, которое позволяет решать широкий круг задач, разрабатывать сложные системы и лучше понимать человеческое поведение, оказалось действенным инструментом в современном мире. Ключевые принципы такого мышления — разбиение сложных задач на более мелкие и управляемые (декомпозиция), распознавание закономерностей и тенденций (выявление закономерностей), абстрагирование характерных особенностей в целях обобщения задачи (абстракция) и создание пошаговых инструкций для решения задач (алгоритмическое мышление).

Вычислительное мышление часто ассоциируется с программированием и информатикой, однако в действительности это базовый навык, применимый ко всем областям исследований, а также к повседневной жизни. Например, в сфере медицины вычислительное мышление может помочь врачам эффективнее диагностировать и лечить заболевания, позволяя разбивать сложные медицинские задачи на более мелкие и управляемые. В мире бизнеса вычислительное мышление может помочь руководителям анализировать данные и выявлять тенденции, которые дают возможность принимать более эффективные решения. Обычным людям вычислительное мышление может помочь решать такие задачи, как организация расписания или составление бюджета.

Поэтому мы должны признать важность вычислительного мышления и его применимость в различных областях. Развивая этот навык, люди могут решать задачи более эффективно и мыслить критически.

1.3.1. Декомпозиция

Декомпозиция — важнейший метод решения задач, предполагающий разбиение сложной задачи или системы на более мелкие и управляемые части, которые можно изучать отдельно. Этот подход полезен тем, что позволяет сосредоточиться на одной части задачи за раз, что гораздо менее трудно, чем пытаться одномоментно решить всю сложную задачу целиком.

Более того, разбив задачу на мелкие и управляемые части, можно легко определить основные причины проблем и выработать эффективные решения, устраняющие их. Кроме того, этот метод позволяет глубже понять задачу или систему, помогая исследовать взаимодействие различных компонентов.

Процесс разбиения задачи на составные части имеет итеративный характер. Он может повторяться на разных уровнях системы, пока программист не получит полное понимание задачи или системы.

Пример

Рассмотрим задачу создания сайта. Она может показаться сложной. Однако если разложить ее на более мелкие части, такие как разработка макета, создание контента, программирование страниц и тестирование сайта, то она станет гораздо более выполнимой. Далее каждую из этих частей также можно раздробить, что еще больше упростит решение задачи.

1.3.2. Выявление закономерностей

Выявление закономерностей — важный когнитивный навык, позволяющий находить сходства или закономерности небольших декомпозированных подзадач, что, в свою очередь, может помочь эффективнее решать более сложные задачи.

На практике выявление закономерностей позволяет идентифицировать ключевые идеи, упущенные из виду другими участниками процесса, и принимать более обоснованные решения в различных областях, начиная с научных исследований и заканчивая бизнес-стратегиями.

Пример

Предположим, что в рамках задачи создания сайта вы реализовали макет дизайна для одной страницы и поняли, что тот же макет можно использовать для других страниц, если внести небольшие изменения. Выявление этой закономерности поможет сэкономить время, поскольку вам не придется создавать дизайн каждой страницы с нуля.

1.3.3. Абстрагирование

Абстрагирование играет основополагающую роль при решении задач. Эта концепция предполагает удаление ненужных деталей и сосредоточение внимания на информации, которая необходима для решения рассматриваемой задачи. Абстрагирование используется в разных областях, таких как математика, информатика и инженерия.

В математике абстрагирование позволяет упрощать сложные задачи путем разбиения их на более мелкие и управляемые части. Аналогичным образом в информатике абстрагирование помогает упростить проектирование сложных программных систем, что облегчает их понимание и сопровождение. Кроме того, абстрагирование в значительной степени служит основой для проектирования, поскольку дает инженерам возможность сосредоточиться на важнейших компонентах проекта, что в итоге приводит к более эффективным решениям.

Таким образом, абстрагирование является важнейшим навыком, позволяющим достигать успеха в различных областях, и его освоение может способствовать принятию более эффективных решений.

Пример

При создании сайта не нужно знать, как компьютер или Интернет работают на фундаментальном уровне. Эти детали абстрагированы. Вам нужно лишь сосредоточиться на создании веб-страниц, используя язык программирования и набор инструментов.

1.3.4. Алгоритмическое мышление

Алгоритмическое мышление — важнейший навык, который подразумевает способность разбивать сложные задачи на более мелкие и управляемые компоненты. Поступая так, вы сможете получить четкое представление о возникшей проблеме, которое затем можно использовать для создания пошагового метода ее решения (также известного как алгоритм).

Один из ключевых аспектов алгоритмического мышления — определение порядка шагов решения задачи. Этот процесс требует тщательного рассмотрения требований и ограничений задачи, а также понимания различных подходов, с помощью которых ее можно решить.

Еще один важный аспект алгоритмического мышления — определение способа объединения отдельных шагов для решения задачи. Этот процесс часто предполагает анализ задачи с разных точек зрения и исследование различных компромиссов, которые могут возникнуть при выборе конкретного подхода.

В целом алгоритмическое мышление является важным навыком для любого, кто хочет решать сложные задачи систематически и эффективно. Разбив задачу на более мелкие компоненты и выработав четкое понимание ее требований и ограничений, можно создать эффективный алгоритм, который поможет быстро и точно прийти к решению.

Пример

При создании сайта вы можете создать алгоритм или набор инструкций для реализации веб-страницы. Этот алгоритм может содержать шаги по настройке структуры HTML, добавлению стилей с помощью CSS и интерактивного поведения с помощью JavaScript.

```
<!DOCTYPE html>
<html>
<head>
    <title>My Website</title>
    <style>
        body{
            background-color: lightblue;
        }
    </style>
</head>
<body>
    <h1>Welcome to My Website</h1>
    <p>This is a sample webpage.</p>
</body>
</html>
```

Этот простой HTML-документ, созданный с помощью алгоритмического процесса, образует базовую веб-страницу.

1.3.5. Отладка и итерация

Вычислительное мышление — непрерывный процесс, который не заканчивается после того, как решение найдено. Оно продолжает играть важнейшую роль, когда мы тестируем решение, выявляем любые ошибки и совершенствуем подходы. Этот итеративный процесс подразумевает такие действия, как разбиение задачи на более мелкие части, выявление закономерностей и абстрагирование деталей в целях создания работающего решения.

Отладка — естественная и важная часть решения задач. Когда алгоритм не дает ожидаемого результата, это событие следует рассматривать как возможность учиться и совершенствоваться. Отладка требует тех же навыков вычислительного мышления, что и разработка алгоритма.

Тщательно исследуя код и разбивая его на более мелкие части, можно определить причины проблем и лучший способ их устранения. Этот процесс тестирования и доработки решения не только помогает создавать более эффективные алгоритмы, но и позволяет совершенствовать навыки вычислительного мышления.

Пример

Если сайт отображается не так, как вы хотите, то следует начать отладку проблемы, определив возможные причины. Где кроется проблема? В HTML, CSS или JavaScript? Существуют ли некие закономерности, которые могут указывать на причины проблем, например, определенные типы содержимого постоянно отображаются неправильно? Какие детали имеют отношение к этой проблеме и какие из них можно абстрагировать?

Выявив и устранив проблему, вы можете заметить, что решение имеет более широкое применение. Это приведет к пересмотру (или итерации) исходного алгоритма. В этом и заключается процесс итерации — в уточнении и улучшении решения на основе полученных результатов и тестирования.

Итеративное уточнение — базовая часть вычислительного мышления и разработки алгоритмов. Первое решение очень редко оказывается лучшим. Поэтому, постоянно совершенствуя наши решения, мы можем создавать более эффективные, более надежные и более адаптируемые алгоритмы.

Из вышесказанного следует, что вычислительное мышление — это инструмент, позволяющий не только находить решения, но и уточнять и улучшать их. Поэтому не бойтесь совершать ошибки и не расстраивайтесь, если что-то идет не так, как ожидалось. С каждой ошибкой и итерацией вы на шаг ближе к лучшему решению.

Погружаясь в мир алгоритмов, помните: речь идет не только о поиске правильного ответа, но и о понимании задачи, изучении различных решений, а также о постоянном обучении и совершенствовании. И самое главное — не забывайте получать удовольствие от процесса!

Говоря о вычислительном мышлении, мы не имеем в виду, что вы должны мыслить как компьютер. Скорее речь идет о применении стратегий, упрощающих использование компьютера для решения задач и позволяющих задействовать для этого его вычислительные возможности. Когда мы изучаем алгоритмы и углубляемся в информатику, четыре элемента вычислительного мышления, которые мы описали выше, служат руководящими принципами.

В следующих главах мы будем часто прибегать к вычислительному мышлению, изучая различные алгоритмы, анализируя их эффективность и решая с их помощью задачи. А пока уделите некоторое время размышлениям

об этих принципах и подумайте, как вы могли бы применить их в своей повседневной жизни.

Помните: как и любой другой навык, вычислительное мышление требует практики. Поэтому не расстраивайтесь, если поначалу его освоение покажется трудным. Продолжайте в том же духе — и постепенно начнете видеть проблемы в совершенно ином свете.

1.4. Практические задачи

Теперь, когда вы познакомились с понятиями «алгоритм» и «вычислительное мышление», пришло время рассмотреть примеры их использования на практике. Цель задач, представленных в этом разделе, — помочь вам освоить вычислительное мышление и дать возможность применить навыки декомпозиции, выявления закономерностей, абстрагирования и алгоритмического мышления. Не волнуйтесь, если у вас не получится решить задачу с первой попытки. Помните, что вычислительное мышление подразумевает итеративное совершенствование подходов.

Задача 1. Генератор паролей

Описание задачи

Напишите алгоритм генерации пароля длины *n*. Пароль должен генерироваться случайным образом и содержать как минимум одну заглавную букву, одну строчную, одну цифру и один специальный символ.

Подсказка

Разбейте задачу на составные части (генерирование заглавной буквы, строчной, цифры, специального символа), а затем объедините их.

Решение

Для решения этой задачи можно создать четыре функции, генерирующие случайную заглавную букву, случайную строчную, случайную цифру и случайный специальный символ соответственно, а затем объединить эти случайно сгенерированные символы и заполнить остальную часть пароля случайными символами из всех групп.

Вот как можно реализовать такое решение на Python:

```
import random
import string

def generate_password(n):
    # n должно быть не меньше 4, чтобы в сгенерированный пароль можно было
    # добавить хотя бы один символ из каждой категории

    assert(n >= 4), "Пароль должен содержать не менее 4 символов"

    password = []
    password.append(random.choice(string.ascii_uppercase))
    password.append(random.choice(string.ascii_lowercase))
    password.append(random.choice(string.digits))
    password.append(random.choice(string.punctuation))

    for i in range(n - 4):
        random_choice = random.choice({0, 1, 2, 3])
        if random_choice == 0:
            password.append(random.choice(string.ascii_uppercase))
        elif random_choice == 1:
            password.append(random.choice(string.ascii_lowercase))
        elif random_choice == 2:
            password.append(random.choice(string.digits))
        else:
            password.append(random.choice(string.punctuation))

    random.shuffle(password)

    return ''.join(password)
```

Задача 2. Календарь событий

Описание задачи

Есть список событий с указанием времени их начала и окончания (в 24-часовом формате). Напишите алгоритм, определяющий, совпадают ли какие-либо события.

Подсказка

Поищите закономерности в пересечении событий и подумайте, как можно абстрагировать задачу, чтобы сделать ее более управляемой.

Решение

Отсортируем список событий по времени начала. Затем пройдемся по этому списку и сравним время окончания текущего события со временем начала следующего. Если время окончания больше времени начала, значит, имеет место перекрытие.

```
def has_overlap(events):
    # Сортировать события по времени начала
    events.sort(key = lambda x: x['start'])

    for i in range(len(events) - 1):
        if events[i]['end'] > events[i + 1]['start']:
            return True

    return False
```

Задача 3. Построение пирамиды

Описание задачи

Напишите алгоритм вывода пирамиды из звездочек с n уровнями. Например, если $n = 3$, то результат должен выглядеть так:

```
  *
 ***
*****
```

Подсказка

Разделите задачу на две части: генерирование каждого следующего уровня пирамиды и объединение созданных уровней. И снова постарайтесь выявить закономерности в количестве звездочек и пробелов.

Решение

Эту задачу можно решить, последовательно выводя уровни пирамиды друг за другом. На каждом уровне выводится сначала несколько пробелов, затем несколько звездочек, а затем снова несколько пробелов.

```
def print_pyramid(n):
    for i in range(n):
        print(' ' * (n - i - 1) + '*' * (2 * i + 1) +' ' * (n - i - 1))
```

Задача 4. Сжатие текста

Описание задачи

Напишите алгоритм сжатия заданной строки путем замены последовательностей одинаковых символов одним символом и числом, определяющим количество его повторений. Например, строку `aaabbbbcc` следует сжать в `a3b4c2`.

Подсказка

Эту задачу можно решить, применив алгоритмическое мышление: разработайте пошаговый процесс перемещения по строке и отслеживания количества вхождений каждого символа.

Решение

Решить эту задачу можно, перебирая строку и запоминая текущий символ и количество его вхождений. Если текущий символ отличается от предыдущего, то добавляем предыдущий символ и его счетчик в результат и сбрасываем счетчик.

```
def compress_text(text):
    compressed = ''
    count = 1

    for i in range(1, len(text)):
        if text[i] == text[i - 1]:
            count += 1
        else:
            compressed += text[i - 1] + str(count)
            count = 1

    # Добавить в результат последний символ и его счетчик
    compressed += text[-1] + str(count)

    return compressed
```

Помните, что эти задачи направлены не только на поиск правильного ответа, но и на улучшение понимания принципов вычислительного мышления и его применения. Сделав попытку решить задачу, подумайте, как вы использовали декомпозицию, выявление закономерностей, абстрагирование и алгоритмическое мышление. Если вы нашли решение, то подумайте, как его усовершенствовать, чтобы сделать более эффективным и элегантным.

Резюме

В этой главе началось наше путешествие по увлекательному миру алгоритмов. Мы определили алгоритм как точную последовательность инструкций, выполнение которых приводит к решению задачи. Затем увидели, как это определение применяется к различным аспектам нашей жизни: начиная с кулинарных рецептов и заканчивая GPS-навигацией и, конечно же, информатикой.

Далее мы исследовали важность алгоритмов в информатике. Они лежат в основе работы компьютеров, позволяют решать сложные задачи, делать выводы и создавать программные приложения. Хорошо продуманный алгоритм может значительно улучшить производительность компьютера.

Затем мы углубились в ключевые принципы вычислительного мышления: декомпозицию, выявление закономерностей, абстрагирование и алгоритмическое мышление. Эти принципы применимы не только к информатике; они могут помочь решать задачи и делать выводы в повседневной жизни.

- **Декомпозиция** представляет собой разбиение сложной проблемы на более простые и управляемые части.
- **Выявление закономерностей** подразумевает нахождение тенденций и сходств, которые могут помочь решить задачу.
- **Абстрагирование** — это процесс сосредоточения внимания на существенных деталях и игнорирования ненужной информации.
- **Алгоритмическое мышление** подразумевает создание пошагового плана решения проблемы или выполнения задачи.

Мы выяснили, что практиковать вычислительное мышление вовсе не означает мыслить как компьютер. Речь идет о применении стратегий, упрощающих использование компьютера для решения задач. Мы также обсудили итеративный характер решения проблем, обучение на основе отладки и совершенствование решений.

В конце главы были приведены практические задачи, призванные помочь вам лучше понять суть вычислительного мышления. Чтобы решить каждую задачу, вы применяли принципы декомпозиции, выявления закономерностей, абстрагирования и алгоритмического мышления.

Помните: чтобы использовать вычислительное мышление, нужно практиковаться. Этот навык можно развивать и оттачивать с течением времени, и он может пригодиться не только в информатике.

В следующих главах мы углубимся в различные типы алгоритмов, проанализируем их сложность и области применения. Поэтому продолжайте читать, практиковаться и наслаждаться путешествием по миру алгоритмов.

Глава 2
ПСЕВДОКОД И БЛОК-СХЕМЫ

https://t.me/it_boooks/2

В этой главе мы рассмотрим важные инструменты, с помощью которых информатики взаимодействуют с алгоритмами и описывают их: псевдокод и блок-схемы. Эти инструменты позволяют представить логику алгоритма в визуальном и текстовом виде, упрощая разбиение сложных задач на более простые шаги.

Псевдокод и блок-схемы особенно полезны для создания и изучения алгоритма, так как позволяют представить его в более простом виде, чем полноценный язык программирования. Псевдокод дает возможность выразить логику алгоритма с помощью простого и понятного языка, имитирующего структуру языка программирования. Блок-схемы, в свою очередь, помогают представить алгоритм в виде диаграммы, которая описывает ход его выполнения и наглядно демонстрирует последовательность шагов и точки принятия решений.

Научившись писать псевдокод и рисовать блок-схемы, вы сможете создавать алгоритмы еще эффективнее. Кроме того, эти два инструмента помогут разобраться в логике, лежащей в основе успешного алгоритма, что может пригодиться при оптимизации или отладке программного кода.

К концу этой главы у вас будет достаточно знаний, позволяющих работать с этими важными инструментами, которые ученые-компьютерщики используют при взаимодействии со своими алгоритмами и их документировании.

Начнем с псевдокода.

2.1. Псевдокод

Псевдокод — ценный инструмент для разработки и представления алгоритмов в понятном для людей виде. В отличие от реального программного кода, псевдокод записывается на естественном языке (в этой книге — на английском), что делает его понятным для людей с небольшим опытом программирования или вообще не имеющим его.

При создании псевдокода основное внимание уделяется логике алгоритма, а не синтаксису конкретного языка программирования. Это означает, что детали конкретного языка не так важны. В псевдокоде используются управляющие конструкции и команды, общие для большинства языков программирования, такие как циклы (`for`, `while`), условные операторы (`if`, `else`) и операторы (`print`, `return`).

Используя псевдокод, люди могут разбивать сложные алгоритмы на более мелкие и управляемые части и затем совершенствовать и оптимизировать каждую из них, прежде чем собирать все воедино. Этот процесс помогает гарантировать надежную и безошибочную работу алгоритма.

Написание псевдокода — отличный способ продумать и разработать алгоритм до его фактической реализации. Он позволяет видеть общую картину и одновременно уточнять детали, что способствует созданию более эффективного и полезного конечного продукта.

Рассмотрим простую задачу: написать алгоритм вычисления суммы чисел в списке. Представить его в псевдокоде можно следующим образом.

```
Алгоритм: сумма чисел в списке
Вход: список чисел — List
Выход: сумма всех чисел в списке — Sum

1. Set Sum to 0
2. For each Number in the List, do
3.     Add Number to Sum
4. End For
5. Return Sum
```

Этот псевдокод смогут понять даже те, кто не знает какого-либо конкретного языка программирования. Здесь мы присваиваем переменной `Sum` значение `0`, а затем перебираем числа в списке, прибавляя их к нашей сумме. В конце мы возвращаем сумму.

С помощью вложенных условий и циклов псевдокод может представлять более сложную логику. Рассмотрим другой пример: алгоритм поиска наибольшего числа в списке.

```
Алгоритм: поиск наибольшего числа
Вход: список чисел — List
Выход: наибольшее число в списке — Largest
```

```
1. Set Largest to the first number in the List
2. For each Number in the List, do
3.     If Number is larger than Largest, then
4.         Set Largest to Number
5.     End If
6. End For
7. Return Largest
```

В этом псевдокоде мы присваиваем переменной `Largest` первое число из списка. Затем проверяем каждое последующее число, и если оно больше текущего значения `Largest`, то обновляем переменную, присваивая ей это новое число. В конце мы возвращаем найденное наибольшее число.

Псевдокод не предназначен для выполнения на компьютере — он служит лишь инструментом для разработки алгоритмов, описания программ перед реализацией и обмена идеями. Им могут пользоваться как программисты, так и люди, не имеющие навыков создания программ.

Вот несколько рекомендаций по написанию псевдокода.

1. **Используйте понятные и описательные имена.** Рекомендуем давать осмысленные имена переменным, процедурам и структурам данных. Например, выбирая имя для переменной, учитывайте, что она делает и как используется в программе. Аналогично, выбирая имя для процедуры, учитывайте, что она делает и какие параметры принимает. Используя понятные и описательные имена, можно улучшить читабельность псевдокода и упростить его последующее преобразование в реальную программу.
2. **Используйте отступы и структуру.** Правильно выставляя отступы и структурируя псевдокод, можно сделать его более понятным и визуализировать ход выполнения программы, особенно при работе с вложенными циклами или условными операторами. Кроме того, отступы и структура сделают ваш код более организованным и простым для чтения и тем самым облегчат его отладку и сопровождение в будущем. Понятный и короткий псевдокод может помочь вам и другим разработчикам понять логику вашей программы и упростить совместную работу и внесение изменений в случае необходимости.
3. **Будьте проще.** Цель псевдокода — упростить сложные структуры программирования до формы, которую легко понять. Поэтому старайтесь не использовать узкоспециализированную терминологию или слишком

большое количество деталей. Сосредоточьтесь на передаче логики алгоритма понятным и коротким способом. Для этого можно, например, разбить алгоритм на более мелкие и управляемые шаги, а затем каждый из них описать простыми словами. Кроме того, важно помнить, что псевдокод не является компилируемым кодом — он позволяет обрисовать базовую структуру алгоритма. Поэтому не беспокойтесь о синтаксисе или форматировании и постарайтесь изложить логику алгоритма на бумаге так, чтобы ее было легко понять.

4. **Будьте последовательны.** В псевдокоде нет жестких правил; тем не менее важно быть последовательным в том, как вы представляете свою логику. Если вы решите использовать определенный формат или стиль для одной части псевдокода, то используйте их и для другой. Это не даст запутаться всем, кто читает ваш псевдокод. Более того, последовательность в оформлении псевдокода может помочь вам выявить ошибки или несоответствия в собственном мышлении. Тот же формат или стиль позволят вам заметить, где вы допустили ошибку или где ваша логика не сходится. Вдобавок последовательность в оформлении псевдокода может помочь другим программистам, работающим с вашим псевдокодом или использующим его в качестве справки. Чем легче им будет понять, как вы структурировали логику, и чем быстрее они смогут найти нужную информацию, тем выше вероятность, что они смогут эффективно задействовать ваш псевдокод.
5. **Добавляйте комментарии.** Они необязательны, но позволяют сделать псевдокод более понятным, особенно когда дело касается сложных разделов. Вдобавок комментарии могут помочь вспомнить ход ваших мыслей при написании псевдокода, когда вы вернетесь к нему спустя какое-то время. Кроме того, благодаря им другие программисты смогут лучше понять ваш псевдокод, что очень полезно при работе над групповыми проектами. И к тому же с помощью комментариев можно объяснить, почему принято определенное решение или выбран тот или иной подход, и благодаря этому все, кто просматривает ваш псевдокод, смогут получить ценную информацию. Проще говоря, если вы потратите время на добавление комментариев, это поможет вам писать более качественный код и укрепить сотрудничество с другими людьми.

Отметим еще раз, что цель псевдокода — прояснить логику и структуру алгоритма, поэтому не существует универсального подхода к его написанию. Главное — пишите код так, чтобы он был понятен вам и всем, кому может понадобиться вникнуть в суть вашего алгоритма.

2.1.1. Гибкость псевдокода

Псевдокод — невероятно универсальный и адаптируемый инструмент, который можно использовать в самых разных контекстах. В отличие от конкретных языков программирования, псевдокод не должен следовать каким-либо определенным правилам. Поэтому он позволяет создавать описание логики и шагов алгоритма, понятное другим людям, даже тем, кто не имеет опыта работы с языками программирования, которые вы используете.

Важно помнить, что основная цель псевдокода — передача главной идеи и логики алгоритма, а не его точная реализация. По сути, псевдокод действует как мост между постановкой задачи и окончательной реализацией в программном коде. Его цель — показать, что нужно, а не как это делается, не конкретные шаги, предпринимаемые для достижения цели при использовании определенного языка программирования.

Благодаря псевдокоду можно гарантировать, что все участники процесса разработки будут понимать общую логику и цели алгоритма. А это позволит всем настроиться на одну волну и может помочь выявить потенциальные проблемы или ошибки до того, как они станут трудноразрешимыми. В целом псевдокод — невероятно полезный инструмент, помогающий упростить процесс разработки и создать более эффективный программный код.

Рассмотрим пример, иллюстрирующий вышесказанное.

Предположим, вы получили задание — создать алгоритм, проверяющий четность числа. Вот как эту проверку можно выразить в виде псевдокода:

```
Алгоритм: проверка четности числа
Вход: число — Num
Выход: "Even", если число четное, "Odd", если нет

1. If Num modulo 2 is equal to 0, then
2.     Print "Even"
3. Else
4.     Print "Odd"
5. End If
```

Этот псевдокод прост и передает суть, не позволяя увязнуть в синтаксисе какого-либо конкретного языка программирования. Важная особенность псевдокода — он четко передает намерения и логику, что и является его конечной целью.

Обратите внимание: разные люди могут писать псевдокод по-своему, и это нормально. В центре внимания всегда должна быть ясность и простота понимания.

2.2. Блок-схемы

Итак, псевдокод — это текстовое описание алгоритма, помогающее понять сложные операции. Он позволяет представить шаги алгоритма, однако иногда бывает сложно визуализировать фактическую последовательность этих операций.

В таких случаях на помощь приходят *блок-схемы*: они позволяют представить логический поток и процесс принятия решений, лежащие в основе алгоритма или программы, в графическом виде. Фигуры и символы используются для описания различных операций и решений, поэтому с помощью блок-схем можно понять даже самые сложные алгоритмы.

Кроме того, блок-схемы могут служить инструментом, позволяющим передавать сложные идеи другим людям, поскольку предлагают наглядный способ отображения логического потока процесса. То есть блок-схемы могут быть полезны всем, кто занимается разработкой или анализом алгоритмов и программ.

Для обозначения различных типов действий или шагов процесса в блок-схемах используются стандартизированные символы. Ниже перечислены те из них, которые применяются наиболее часто.

- **Овал или прямоугольник со скругленными углами.** Этот символ обычно используется для обозначения начала или конца программы либо процесса, состоящего из нескольких шагов или этапов.

  Например, в проекте разработки программного обеспечения этот символ может обозначать начало этапа планирования проекта, решающего такие задачи, как сбор требований, создание плана проекта и распределение ресурсов. Аналогично символ может обозначать завершение проекта, в том числе такие задачи, как тестирование, развертывание и сопровождение.

- **Прямоугольник.** Применяется для обозначения действия или этапа обработки, например вычисления выражения или выполнения операции.

  В блок-схемах этот символ может указывать на вычисления, операции или любую другую деятельность, связанную с обработкой информации. Прямоугольники помогают визуализировать и понять различные этапы процесса, что делает их символом, облегчающим выбор вариантов решения задач.

- **Ромб.** Используется в программировании для обозначения точки принятия решения, например условного оператора или цикла. Точка принятия

решения — это критический момент в потоке выполнения программы, когда необходимо сделать выбор между двумя или несколькими возможными путями.

Обычно ромб используется для обозначения условного оператора, такого как `if-else`, где программа должна выбрать между двумя разными блоками кода, исходя из результата логической проверки. С помощью ромба также можно представлять оператор цикла, в котором программа повторяет набор инструкций до тех пор, пока не выполнится определенное условие.

Используя символ ромба, программисты могут структурировать ход программы и сделать ее более эффективной.

- **Стрелки.** Эти символы играют очень важную роль при создании блок-схем, поскольку позволяют создать визуальное представление последовательности действий. Стрелки, соединяющие другие символы, помогают показать, как движется процесс и как каждый шаг зависит от предыдущего.

  Кроме того, с помощью стрелок можно отображать различные типы потоков, таких как материальный поток, поток информации или решений, что позволяет выявлять потенциальные проблемы или неэффективность процесса.

  Поэтому очень важно при создании блок-схем процессов использовать стрелки правильно, чтобы они точно отображали процесс и были понятны другим участникам.

Проиллюстрируем блок-схему на примере. Рассмотрим алгоритм, определяющий, является ли число положительным, отрицательным или нулевым. Вот как можно представить такой алгоритм в виде блок-схемы:

```
Начало
   |
   v
Number > 0 --Да--> Вывести "Положительное"
   |
   Нет
   v
Number < 0 --Да--> Вывести "Отрицательное"
   |
   Нет
   v
Вывести "Ноль"
   |
Конец
```

На этой блок-схеме мы начинаем сверху и следуем по стрелкам в зависимости от принимаемых решений. Если число `Number` больше нуля, то мы выводим `"Положительное"`. Если нет, то переходим к следующей точке принятия решения: если число меньше нуля, мы выводим `"Отрицательное"`. Если оно не больше и не меньше нуля, то, значит, оно равно нулю, поэтому мы выводим `"Ноль"`.

Как и псевдокод, блок-схемы не предназначены для выполнения на компьютере. Они служат инструментом планирования, проектирования и визуализации алгоритмов, упрощая взаимодействие с другими людьми и обсуждение с ними логики алгоритма, в том числе с теми, кто не имеет большого опыта в программировании.

Блок-схемы особенно полезны при работе со сложными алгоритмами, так как могут прояснить логику выполнения и облегчить выявление потенциальных ошибок или неэффективных конструкций алгоритма.

Вот несколько рекомендаций, которые следует учитывать при создании блок-схем.

- **Упрощайте.** Одно из преимуществ блок-схемы — они помогают понять процесс. При разбиении сложных шагов на более мелкие и управляемые этапы блок-схема может облегчить даже самый трудный процесс.

  Поэтому важно, чтобы блок-схемы были максимально простыми. Это означает, что вы должны удалить любые ненужные детали или шаги, которые могут запутать читателя, и сосредоточить внимание только на ключевых идеях, которые необходимо донести.

- **Используйте стрелки последовательно.** Обычно стрелки в блок-схемах идут сверху вниз и слева направо. Последовательность направления помогает избежать путаницы. При создании блок-схем важно сохранять единообразие стрелок. Таким образом вы поможете избежать путаницы и понять вашу блок-схему.

  Кроме того, рекомендуется использовать стрелки для обозначения направления потока информации. Это может помочь направить взгляд читателя в определенное место и облегчить отслеживание потока. Можно также добавлять метки и подписи, объясняющие различные этапы блок-схемы.

  Это особенно полезно, если блок-схема сложна или вам нужно передать большой объем сведений. Потратив время на создание четкой и последовательной блок-схемы, вы сможете гарантировать, что ваша аудитория поймет информацию, которую вы пытаетесь донести.

- **Подписывайте стрелки.** При создании блок-схемы важно подписывать стрелки, особенно когда она разветвляется в точках принятия решений. Это позволит показать, при каких условиях выполнение пойдет по тому или иному пути.

  В результате блок-схема становится более организованной, понятной и эффективной. Никогда не забывайте подписывать стрелки — и ваша блок-схема будет понятной и точной.

- **Четко определяйте каждый шаг.** Каждый шаг в блок-схеме, представляемый в виде отдельного прямоугольника, должен быть атомарным, то есть его нельзя разбить на более мелкие этапы. Это отдельная операция, выполняющая конкретное действие.

  Чтобы сделать блок-схему более понятной, можно подробно описать действия, происходящие на каждом шаге. Для этого можно добавить описательный текст, в котором подробно объясняется каждый из этапов.

  Описывая четкие и краткие шаги в блок-схеме, вы можете быть уверены, что она точно отображает процесс.

- **Проверяйте свои блок-схемы.** Закончив составлять блок-схему, обязательно просмотрите ее и убедитесь, что она точно отображает процесс, который вы пытаетесь показать. Это не только поможет вам выявить недостающие шаги или логические ошибки, но и даст возможность обнаружить любые потенциальные улучшения, которые можно внести в процесс.

  Например, вы можете выявить возможность объединения определенных этапов, вообще их исключить либо использовать технологии или автоматизацию для оптимизации процесса. Потратив время на проверку блок-схемы, вы будете уверены, что она не только точно отображает текущее состояние процесса, но и позволяет вносить постоянные улучшения.

- **Выбирайте правильный уровень детализации.** При создании блок-схемы важно учитывать уровень детализации, оптимальный для эффективной передачи информации. Если он слишком высокий, то блок-схема может стать запутанной и трудной для понимания. А если слишком низкий, то она может оказаться малополезной из-за недостатка информации.

  Определяя оптимальный уровень детализации, важно учитывать сложность алгоритма и целевую аудиторию блок-схемы. Если алгоритм очень сложен, то может понадобиться добавить более подробную информацию, которая облегчит понимание блок-схемы. Однако если целевая аудитория не знакома с темой, то дополнительную информацию лучше добавлять для того, чтобы предоставить людям больше сведений о ситуации.

Важно учитывать и назначение блок-схемы. Если ее цель — дать общий обзор алгоритма, то целесообразнее использовать более низкий уровень детализации. Однако если блок-схема должна играть роль пошагового руководства, то уровень может быть более высоким.

Таким образом, при создании блок-схемы важно найти баланс между предоставлением достаточного количества деталей и обобщенностью, чтобы схема была полезной и не перегружала читателя. Этот баланс часто зависит от сложности алгоритма и целевой аудитории.

- **Разбивайте сложные процессы на части.** При работе со сложным алгоритмом нередко полезно разбить процесс на более мелкие и управляемые подпроцессы. Это поможет лучше понять и визуализировать каждый этап.

  Любой подпроцесс можно представить в виде отдельной блок-схемы, что позволит сосредоточиться на каждой части алгоритма. Получив представление обо всех подпроцессах, можно связать их и создать обобщенную блок-схему, наглядно демонстрирующую весь алгоритм.

  Так вам будет проще объяснить алгоритм другим людям либо выявить потенциальные ошибки или неэффективные шаги.

- **Используйте цвета и элементы дизайна.** Это необязательно, тем не менее с помощью цвета и других элементов дизайна можно сделать блок-схему более понятной и привлекательной. Разные цвета могут использоваться для обозначения разных типов операций, выделения наиболее важных частей алгоритма или маркировки потока данных.

  Например, один цвет можно использовать для обозначения точек принятия решения, другой — операций ввода-вывода, третий — расчетов. Это визуальное разделение особенно полезно в случае, если с блок-схемами работают люди, плохо знакомые с ними или алгоритмами. Кроме того, цвет позволяет разбить большие блоки текста на более мелкие и сделать блок-схему более эстетичной.

  Элементы дизайна, такие как стрелки, прямоугольники и линии, тоже могут помочь сделать блок-схему более структурированной и простой для понимания. Например, стрелки могут указывать направление потока данных, а прямоугольники можно использовать для группировки взаимосвязанных операций или действий.

- **Пересматривайте и уточняйте.** Блок-схема помогает понять структуру логического потока алгоритма. Однако, как и при использовании любого другого инструмента, первый вариант блок-схемы вряд ли получится

идеальным. Поэтому важно пересматривать ее и вносить в нее уточнения по мере разработки алгоритма.

В ходе пересмотра вы сможете выявить любые ошибки, несоответствия или неэффективность вашего алгоритма и внести необходимые коррективы. Более того, по мере появления новых идей или изменений в процессе разработки вы можете соответствующим образом изменить блок-схему, чтобы она точно представляла обновленную версию вашего алгоритма.

Этот итеративный процесс пересмотра и уточнения блок-схемы в итоге приведет к созданию более надежного и эффективного алгоритма, позволяющего достичь желаемых результатов.

- **Используйте программное обеспечение для составления блок-схем.** Существует множество программных инструментов, которые помогут вам проектировать и визуализировать ваши идеи и создавать блок-схемы профессионального качества. Некоторые из инструментов позволяют перетаскивать символы; добавлять в блок-схему элементы и изменять их; выполнять автоматическое выравнивание и позиционирование, благодаря чему блок-схемы становятся аккуратными и структурированными; использовать предустановленные цветовые схемы, которые помогают сделать схему более привлекательной и повысить ее читабельность, и т. д.

  Все эти специализированные инструменты могут оказаться хорошим подспорьем, если вы создаете сложные блок-схемы или делитесь ими с другими людьми. Такие программы часто содержат функции, позволяющие выполнять совместную работу, экспортировать схемы в различные форматы или отправлять их другим участникам процесса по электронной почте или через облачные платформы. Эти функции могут облегчить совместную работу над сложными проектами и гарантировать взаимопонимание.

  В целом такое программное обеспечение может стать отличной инвестицией для тех, кому часто приходится рисовать блок-схемы или участвовать в коллективной работе над сложными проектами. Благодаря широкому спектру функций и преимуществ эти инструменты обязательно помогут улучшить рабочий процесс и качество вашей работы.

Для представления сложных алгоритмов в блок-схемах используется множество дополнительных символов, обозначающих различные операции, такие как ввод-вывод данных, подготовка данных, ручные операции и т. д.

Однако для того, чтобы сохранить простоту и сосредоточенность на основной логике алгоритмов, обычно применяются только основные символы, перечисленные выше.

Подводя итог, можно сказать, что блок-схемы позволяют визуализировать логику алгоритма и передавать ее другим. Вместе с псевдокодом они составляют базовый набор инструментов, служащих для разработки и документирования алгоритмов.

2.3. Выражение реальных задач в виде псевдокода

Одним из наиболее важных навыков, которыми должен обладать программист или аналитик данных, является способность разбить реальную задачу на более мелкие компоненты и сформулировать решение, которое может выполнить компьютер, — алгоритм.

Для этого необходимо совершить промежуточный шаг — написать псевдокод. Тот, в свою очередь, позволяет подать логику алгоритма в понятных людям терминах, прежде чем переводить ее в синтаксис конкретного языка программирования. Этот шаг важен для разработки эффективных алгоритмов, поскольку помогает тщательно спланировать их и протестировать перед программированием.

Кроме того, псевдокод можно использовать как инструмент совместной работы, поскольку он дает возможность четко и ясно донести ваши идеи до других членов команды или заинтересованных сторон. В целом навык использования псевдокода очень ценный и может значительно улучшить процесс разработки и качество конечного продукта.

Далее описываются общие этапы процесса, которому вы можете следовать, столкнувшись с реальной задачей.

- **Изучение задачи.** Первый и самый важный шаг — понять задачу, которую вы должны решить. Какие входные данные и какой результат вы ожидаете получить? Какие ограничения и особые условия необходимо учитывать?

  Эффективно решить стоящую перед вами задачу можно при условии, что затраченных времени и усилий достаточно для того, чтобы досконально разобраться в ней. Под этим подразумевается тщательный анализ ожидаемых входных и выходных данных, а также любых ограничений или особых условий, которые могут повлиять на решение. Возможно, вам

придется проконсультироваться с заинтересованными сторонами или экспертами в предметной области, чтобы получить более полное представление о задаче.

Кроме того, важно разбить задачу на более мелкие и управляемые компоненты, чтобы выявить любые потенциальные сложности. Потратив время на изучение задачи, вы сможете разработать более эффективное решение и в конечном счете достичь желаемого результата.

- **Декомпозиция задачи.** Разбейте ее на более мелкие и управляемые части. Каждая часть должна быть меньшей по размеру, чем общая, и способствовать ее решению.

  Выполнив декомпозицию, вы сможете сосредоточиться на каждой отдельной части и работать над решением общей задачи более планомерно. Вы будете понимать, что необходимо сделать и как это помогает общему решению.

  Такой подход позволяет быть уверенным в том, что каждая часть задачи получит столько внимания, сколько нужно для ее эффективного решения, и ни один шаг в этом процессе не будет упущен из виду или проигнорирован.

- **Проектирование алгоритма.** Выделив подзадачу, определите шаги, которые позволят ее решить (можно разбить каждую подзадачу на еще более мелкие), и расположите их в логической последовательности. Определив все шаги, необходимые для каждой подзадачи, вы сможете объединить их и сформировать единый алгоритм.

  Он должен быть понятным, коротким и простым. Помните, что алгоритм — это просто последовательность шагов, предназначенных для решения конкретной задачи, поэтому убедитесь, что ваш алгоритм является комплексным и содержит все необходимые шаги. Благодаря тщательному проектированию алгоритма вы будете уверены, что сможете эффективно решить каждую подзадачу, сведя к минимуму риск ошибок на этом пути.

- **Разработка псевдокода.** Получив алгоритм, вы сможете записать его в псевдокоде. Используйте простой описательный язык для определения каждого шага, применяя стандартные нотации для представления точек принятия решения, циклов и других структур управления.

  В псевдокоде содержится простое описание шагов, иллюстрирующее реализацию алгоритма. Его можно использовать в качестве образца и легко преобразовать в реальный программный код на любом языке программирования, когда придет время реализовать алгоритм.

Рассмотрим простой практический пример. Предположим, вас попросили написать программу, которая вычисляет среднюю оценку учащегося на основе списка оценок.

Решить эту задачу можно так.

- **Шаг 1. Изучение задачи.** Чтобы вычислить среднюю оценку по заданному списку, нужно выполнить простое, но важное действие: суммировать все оценки, учитывая значение каждой из них.

  Этот шаг имеет решающее значение, поскольку гарантирует возможность вычисления среднего значения. Выполнив суммирование, нужно разделить сумму оценок на их общее количество в списке. На этом последнем этапе выводится средняя оценка по списку, которая позволяет понять общую эффективность оцениваемой группы.

- **Шаг 2. Декомпозиция задачи.** Рассматриваемую задачу можно разделить на две меньшие. Первая подзадача предполагает вычисление суммы всех оценок, что может быть утомительным и трудоемким процессом, но тем не менее имеет решающее значение для получения окончательного результата.

  После получения суммы настает черед второй подзадачи, которая вычисляет среднее значение. Следует отметить, что она так же важна, как и первая, поскольку среднее значение является конечным показателем общей успеваемости учащегося. Поэтому важно быть очень осторожным при выполнении обеих подзадач, чтобы получить точный конечный результат.

- **Шаг 3. Проектирование алгоритма.** Основной алгоритм выглядит так:

  1) инициализировать переменную, хранящую общую сумму оценок;
  2) перебрать оценки в списке и прибавить каждую оценку к общей сумме;
  3) разделить общую сумму на количество оценок, чтобы получить среднее значение.

- **Шаг 4. Разработка псевдокода.**

```
Алгоритм: вычисление средней оценки
Вход: список оценок — Grades
Выход: средняя оценка

1. Initialize Total to 0
2. For each Frade in Grades, do
3.     Add Grade to Total
4. End For
5. Set Average to Total divided by the number of Grades
6. Return Average
```

В этом псевдокоде содержится четкое пошаговое описание алгоритма вычисления средней оценки, без учета специфики какого-либо конкретного языка программирования.

Приведем еще несколько рекомендаций, которые следует соблюдать при создании псевдокода.

- **Пишите для своей аудитории.** Разрабатывая псевдокод, важно помнить о том, кто составляет вашу целевую аудиторию. Подумайте, кто будет его читать. Если вы работаете в команде, то его должны понимать все ваши коллеги. Это означает, что вы должны использовать язык и синтаксис, понятные каждому участнику команды. Если вы пишете для себя, то убедитесь, что сможете понять описание, когда вернетесь к нему позже. В противном случае псевдокод будет бесполезным для вас.

  Мы рекомендуем всегда делать ревью псевдокода и пересматривать его, чтобы убедиться, что он прост и понятен как вам, так и другим людям, которым он может понадобиться.

- **Используйте соглашения по именованию.** При написании псевдокода задавайте понятные, описательные имена для переменных и функций. Благодаря этому любой, кто читает ваш псевдокод, легко поймет, что делает каждая часть алгоритма. Это не только упростит работу других людей с вашим кодом, но и облегчит вам его дальнейшую отладку и сопровождение.

  Несоблюдение соглашений по именованию может привести к путанице и ошибкам, особенно при работе над более крупными командными проектами. Поэтому потратьте время и внимательно оцените ваши соглашения: убедитесь, что во всем псевдокоде одни и те же элементы названы одинаково.

- **Учитывайте пограничные случаи.** При проектировании алгоритма важно учитывать пограничные случаи, то есть сценарии, возникающие на «краях» предметного пространства и часто упускаемые из виду. Эти случаи могут оказать существенное влияние на производительность и точность алгоритма. Например, что должен делать алгоритм, если список оценок пуст? В этом случае он должен вернуть соответствующее сообщение и предложить пользователю ввести несколько допустимых оценок.

  Еще один пограничный случай, который следует учитывать, — что происходит, когда список содержит только одну оценку. В этом случае алгоритм все равно должен вычислить среднюю оценку и вернуть содержательный

результат. Кроме того, вам следует подумать о том, что произойдет, если список оценок содержит повторяющиеся значения. Ваш алгоритм должен уметь обрабатывать такие случаи и выдавать точные результаты.

Важно учитывать производительность алгоритма и при работе с большими наборами данных. Если он не оптимизирован для обработки таких данных, то может замедлиться или даже дать сбой. Чтобы этого избежать, следует рассмотреть возможность реализации таких методов, как кэширование или параллельная обработка.

Учитывая такие пограничные случаи и оптимизируя производительность алгоритма, вы будете уверены, что он даст точные результаты в различных сценариях и благодаря этому станет ценным инструментом для решения ваших задач.

- **Комментируйте и объясняйте.** Обязательно добавляйте в псевдокод комментарии и пояснения, особенно при решении сложных задач. Это поможет сделать так, чтобы ваш мыслительный процесс был понятен вам и тем, кому может понадобиться прочитать ваш код.

  Кроме того, комментарии и пояснения могут дать дополнительную информацию для предстоящих изменений или обновлений кода и упростят его сопровождение в долгосрочной перспективе.

- **Возвращайтесь к алгоритму снова и снова.** Не ждите, что ваш псевдокод получится идеальным с первой попытки. Приступив к его реализации в программном коде или получив новую информацию, вы можете столкнуться с необходимостью пересмотреть и уточнить его.

  Важно помнить, что процесс создания псевдокода — это не задача в одно действие. Весьма вероятно, что вам потребуется выполнить несколько итераций доработки по мере реализации кода или при получении новой информации.

  Потратив время на многократный обзор и уточнение псевдокода, в результате вы получите лучший код и более эффективную его реализацию. Так что не расстраивайтесь, если ваша первая попытка не будет идеальной. Воспользуйтесь возможностью повторить и улучшить алгоритм.

Рассмотрим пограничный случай в нашем предыдущем примере:

```
Алгоритм: вычисление средней оценки
Вход: список оценок — Grades
Выход: средняя оценка
```

```
1. If Grades is empty then
2.     Return "Нет оценок для вычисления среднего значения"
3. End If
4. Initialize Total to 0
5. For each Grade in Grades, do
6.     Add Grade to Total
7. End For
8. Set Average to Total divided by the number of Grades
9. Return Average
```

Здесь мы добавили в начало алгоритма дополнительный шаг, обрабатывающий пограничный случай, когда список оценок пуст.

Перевод реальных задач в псевдокод — крайне важный навык при разработке алгоритмов и решении задач в информатике. Практикуясь и развивая его, вы сможете с легкостью разрабатывать алгоритмы, эффективно решающие любую задачу, с которой вам доведется столкнуться.

2.4. Практические задачи

Чтобы действительно освоить написание псевдокода и понять суть блок-схем, важно практиковаться. Ниже приводится несколько задач, которые помогут вам закрепить полученные знания.

Задача 1. Вывод чисел

Разработайте алгоритм, который выводит все числа от 1 до 100, делящиеся на 7.

Решение в псевдокоде:

```
Алгоритм: вывод чисел, кратных 7
Вход: нет
Выход: числа в диапазоне от 1 до 100, кратные 7

1. For each number N from 1 to 100, do
2.     If N is divisible by 7 then
3.         Print N
4.     End If
5. End For
```

Задача 2. Выявление палиндрома

Разработайте алгоритм, проверяющий, является ли палиндромом заданное слово. Палиндром — это слово, которое одинаково читается как в прямом, так и в обратном направлении, например «казак».

Решение в псевдокоде:

```
Алгоритм: проверка палиндрома
Вход: слово — Word
Выход: сообщение "Палиндром"/"Не палиндром"

1. Initialize Start to 0
2. Initialize End to the length of Word - 1
3. While Start is less than End, do
4.    If character at position Start in Word is not equal to the character
at position End in Word then
5.        Return "Не палиндром"
6.    End If
7.    Increment Start by 1
8.    Decrement End by 1
9. End While
10. Return "Палиндром"
```

Задача 3. Поиск наибольшего числа

Разработайте алгоритм, отыскивающий наибольшее число в заданном списке чисел.

Решение в псевдокоде:

```
Алгоритм: поиск наибольшего числа
Вход: список чисел — Numbers
Выход: наибольшее число в списке

1. Initialize Highest to the first number in Numbers
2. For each Number in Numbers, do
3.    If Number is greater than Highest then
4.        Set Highest to Number
5.    End If
6. End For
7. Return Highest
```

Рекомендуем также попробовать нарисовать блок-схемы этих алгоритмов. С помощью этого упражнения вы закрепите знания о том, как алгоритмы

представляются графически, и сможете развить навык работы с обеими формами представления алгоритмов.

Помните: практика — это ключ к обучению. Решая эти задачи и создавая собственные, вы улучшите свою способность превращать реальные задачи в алгоритмы.

Резюме

В этой главе мы рассмотрели основные инструменты, используемые при разработке алгоритмов: псевдокод и блок-схемы. Изучение этих тем помогло нам понять, насколько ясность, простота и точность важны при разработке алгоритмов.

Мы начали наше исследование со знакомства с псевдокодом. Как уже было сказано, псевдокод — это неформальное обобщенное описание компьютерной программы или алгоритма. Он использует структурные соглашения программирования, но предназначен для чтения человеком, а не машиной. Основная цель псевдокода — помочь программистам определить, что программа должна делать и каким образом.

Затем мы перешли к блок-схемам и выяснили, что они представляют собой графическое изображение алгоритма или процесса. В блок-схеме используются различные символы и стрелки, помогающие отобразить шаги и последовательность процесса. Мы подчеркнули, что блок-схемы прекрасно подходят для разработки и исследования даже очень сложных процессов и будут особенно полезны людям, которые лучше воспринимают визуальные материалы.

Исследование привело нас к осознанию важности роли псевдокода и блок-схем в решении задач. Эти представления алгоритмов, хотя и не выполняются компьютерами, позволяют воплощать решения задач реального мира в компьютерные программы. Они представляют собой промежуточный этап на пути к решению задач, дающий возможность сосредоточиться на разработке логики и алгоритмов, а не на отладке синтаксиса и ошибок.

Затем мы поговорили о том, как выразить решение задач в виде псевдокода. Мы отметили основные этапы данного процесса, такие как исследование задачи, разбиение ее на более мелкие части, проектирование алгоритма и, наконец, выражение его в форме псевдокода. Мы также продемонстрировали весь этот процесс на примере вычисления средней оценки по списку оценок.

Чтобы помочь вам закрепить полученные знания, мы завершили главу практическими задачами, начиная с поиска чисел, кратных 7, и заканчивая проверкой того, является ли слово палиндромом, и поиском самого большого числа в списке. С помощью этих задач мы предоставили вам возможность применить новые знания в практических сценариях.

На протяжении главы мы постоянно подчеркивали, что псевдокод и блок-схемы — это инструменты, помогающие мыслительному процессу и решению задач, а не самоцель. Они предназначены в первую очередь для программиста и помогают ему разобраться в задаче, понять ее и найти подходящее решение.

Понятия, которые вы изучили в данной главе, позволят перейти к изучению более сложных алгоритмических структур, речь о которых пойдет далее. Они просты, но имеют огромный потенциал и закладывают основу систематического и логического подхода к исследованию и решению сложных задач.

Помните: ключ к освоению псевдокода и блок-схем — практика. Чем больше задач вы будете решать, тем комфортнее будете себя чувствовать при работе с этими формами представления алгоритмов. В свою очередь, это ощущение будет способствовать вашему развитию как специалиста по решению задач, проектировщика алгоритмов или разработчика программного обеспечения.

Глава 3
ЭФФЕКТИВНОСТЬ АЛГОРИТМОВ

https://t.me/it_boooks/2

Изучая алгоритмы в этой главе, мы будем познавать важность их анализа и оценки эффективности. Это позволит нам принимать обоснованные решения о том, какие алгоритмы использовать в тех или иных сценариях.

Для начала мы познакомимся с фундаментальными понятиями эффективности алгоритмов, в том числе с тем, что такое временна́я и пространственная сложности. Освоив эти понятия, мы сможем определять и сравнивать эффективность различных алгоритмов.

Далее мы рассмотрим различные методы повышения эффективности алгоритмов, такие как мемоизация и динамическое программирование. Мы увидим, как с помощью этих методов, примененных к различным типам алгоритмов, можно достичь оптимальной эффективности.

Мы также обсудим компромиссы между эффективностью алгоритма и другими факторами, такими как простота кода и удобство сопровождения. Понимая эти компромиссы, мы сможем принимать обоснованные решения о выборе алгоритмов в различных сценариях.

Таким образом, в этой главе вы получите инструменты и знания, необходимые для оценки эффективности алгоритмов. Обладая этими знаниями, вы сможете принимать обоснованные решения о том, какие алгоритмы использовать для оптимизации работы и достижения желаемых результатов.

3.1. Временная сложность

Если вам доводилось писать компьютерные программы или разрабатывать алгоритмы, то вы знаете, что всякую задачу можно решить множеством способов. Каждый из них имеет свои достоинства и недостатки. Одни решения работают быстрее, другие — эффективнее, а третьи лучше подходят для определенных типов входных данных. При этом не все решения одинаковы, и некоторые из них могут даже не завершиться за разумное время при работе с большими объемами входных данных.

Именно здесь уместно обратиться к понятию временно́й сложности. Под ней подразумевается количество времени, необходимое для выполнения алгоритма, в зависимости от объема входных данных. Это важный показатель, который следует учитывать при разработке алгоритмов, поскольку он помогает оценить время, необходимое для выполнения алгоритма, и может использоваться для его оптимизации. Понимая, из чего складывается временная сложность алгоритма, вы сможете принимать обоснованные решения о том, какой подход выбрать, и гарантировать эффективную работу вашей программы.

Рассмотрим простой пример: поиск максимального числа в списке целых чисел.

Алгоритм 1

```
def find_max(numbers):
    max_num = numbers[0]
    for number in numbers:
        if number > max_num:
            max_num = number
    return max_num
```

Этот алгоритм просматривает каждое число в списке один раз. Поэтому при наличии в списке n чисел алгоритм выполнит n шагов. Такое соотношение часто называют линейной временной сложностью и обозначают ее как $O(n)$, где n — объем входных данных.

Теперь рассмотрим менее эффективный способ решения той же задачи.

Алгоритм 2

```
def find_max(numbers):
    max_num = numbers[0]
    for i in range(len(numbers)):
        for j in range(i+1, len(numbers)):
            if numbers[j] > max_num:
                max_num = numbers[j]
    return max_num
```

Этот алгоритм для каждого следующего числа просматривает остальную часть списка. В результате количество выполняемых им шагов пропорционально $n*(n - 1) / 2$, которое можно упростить примерно до $(n$^$2) / 2$. Такое соотношение называется квадратичной временной сложностью и обозначается как $O(n$^$2)$.

Несмотря на то что оба алгоритма успешно решают одну и ту же задачу, первый эффективнее, и его эффективность тем очевиднее, чем длиннее

список чисел, поскольку выполнение этого алгоритма требует меньшего количества шагов.

Временная сложность — базовое понятие информатики, особенно важное в области алгоритмов и структур данных. Усвоив его, вы сможете выбирать или разрабатывать наиболее эффективные алгоритмы для конкретных задач. Прочитав эту главу, вы получите более глубокое представление о временной сложности и других факторах, влияющих на эффективность алгоритмов.

3.1.1. Нотация «О большое»

«О большое» — важное понятие в информатике, которое используется для описания верхней границы временной сложности. Эта нотация помогает определить наихудший сценарий — верхний предел времени, затрачиваемого алгоритмом, с точки зрения размера входных данных.

Используя нотацию «О большое», информатики могут оценить эффективность алгоритма и определить, подходит ли он для конкретной задачи. Более того, эта нотация предлагает общий язык, позволяющий информатикам плодотворно обсуждать производительность алгоритмов. В целом нотация «О большое» помогает оптимизировать алгоритмы и разрабатывать эффективные программы.

Еще раз рассмотрим алгоритм поиска максимального числа в списке:

```
def find_max(numbers):
    max_num = numbers[0]
    for number in numbers:
        if number > max_num:
            max_num = number
    return max_num
```

Как уже говорилось выше, этот алгоритм имеет линейную временную сложность, или $O(n)$. Но что это значит? В худшем случае при обработке списка из n чисел алгоритму придется просмотреть n чисел. Даже если максимальное число является первым в списке, алгоритм все равно проверит весь список, чтобы убедиться, что не пропустил большего числа, которое могло находиться дальше в списке. Следовательно, в нотации «О большое» мы выражаем его временную сложность как $O(n)$ — линейное время.

Другой важный аспект, который следует учитывать, — операции с постоянным временем выполнения. В контексте обсуждения временной сложности принято игнорировать константы, поскольку они не меняются и не зависят

от объема входных данных. Например, операция `max_num = numbers[0]` считается операцией с постоянным временем (O(1)), так как всегда выполняется за одно и то же время независимо от размера списка.

Мы еще рассмотрим подробнее «О большое» в разделе 3.3.

3.1.2. Разница между временной сложностью в лучшем, среднем и худшем случаях

При разработке алгоритмов мы обычно фокусируемся на временной сложности наихудшего случая, которая выражается нотацией «О большое», но не следует забывать и о других факторах, которые могут повлиять на производительность алгоритма. Например, временная сложность в среднем случае может быть более реалистичным показателем работы алгоритма на практике, тогда как временная сложность в лучшем случае может дать представление об эффективности алгоритма лишь при определенных условиях.

Важно учитывать и такие факторы, как потребление памяти, объем входных данных и иногда особенности конкретной аппаратной и программной среды, в которой будет работать алгоритм. Получив более полное представление о производительности алгоритма, мы сможем принимать более обоснованные решения о том, когда и как использовать его в реальных приложениях.

Рассмотрим временную сложность в разных случаях более подробно.

- **Временная сложность в лучшем случае** подразумевает сценарий работы алгоритма, когда входные данные находятся в наиболее благоприятном состоянии. Так, для алгоритма сортировки лучшим сценарием будет ситуация, когда входной список уже отсортирован, поскольку на то, чтобы решить задачу, алгоритму потребуется минимальное количество времени и ресурсов. Напротив, худшим сценарием для алгоритма сортировки будет ситуация, когда входной список отсортирован в обратном порядке, в связи с чем алгоритму потребуется выполнить максимальное количество сравнений и перестановок. Временная сложность в лучшем случае — важный показатель, который следует учитывать при оценке эффективности алгоритма, поскольку он дает представление о производительности алгоритма в идеальных условиях.
- **Временная сложность в среднем случае** — это показатель ожидаемого времени, необходимого для решения задачи на основе среднего сценария. Показатель учитывает распределение всех возможных входных данных.

Другими словами, предполагается, что входные данные случайным образом распределены по всем возможным значениям. Определение этого показателя может осложняться трудностью определения того, как выглядят «средние» входные данные. На сложность в среднем случае могут влиять такие факторы, как используемая структура данных, количество элементов во входных данных и тип алгоритма. Важно понимать, что такое средняя сложность алгоритма, поскольку она позволяет более реалистично оценить время, которое требуется алгоритму для решения задачи на практике.

Временная сложность в худшем случае (которую мы уже обсудили) — это показатель максимального времени, необходимого алгоритму для решения данной задачи. Показатель описывает сценарий, в котором алгоритм работает хуже всего и тратит максимальное время на решение задачи. Обычно это самый важный показатель, который следует учитывать при анализе эффективности алгоритма, поскольку он определяет верхнюю границу, хуже которой алгоритм точно не будет работать.

Стоит отметить, что временная сложность в худшем случае не всегда является наиболее репрезентативным показателем производительности алгоритма. В некоторых ситуациях алгоритм может работать намного лучше, чем описывает его сложность в худшем случае. Поэтому стоит иметь в виду и два других типа временной сложности.

При анализе временной сложности алгоритма также важно учитывать объем входных данных. В некоторых ситуациях алгоритм может иметь превосходную временную сложность в худшем случае для небольшого объема входных данных, но плохо работать при увеличении этого объема. В таких ситуациях следует выявить узкое место алгоритма и рассмотреть альтернативные варианты, которые могут лучше справляться с большими объемами входных данных.

Совокупность этих сложностей дает представление о потенциальной производительности алгоритма. Он может иметь одинаковую временную сложность в лучшем, среднем и худшем случаях, или же эти показатели могут существенно различаться. Знать этот диапазон бывает полезно, однако на практике мы чаще сосредоточиваемся на худшем сценарии, чтобы гарантировать производительность наших алгоритмов даже в самых неблагоприятных условиях.

Помните: разбираться во временной сложности и эффективности алгоритма нужно не только для того, чтобы найти или создать наиболее эффективный алгоритм. Понимание этих факторов также позволяет оценить компромиссы

и принять обдуманное решение, основанное на конкретных потребностях программного обеспечения или приложения.

В следующих разделах мы углубимся в исследование различных классов временной сложности (например, постоянное время O(1), логарифмическое время O(log *n*), линейное O(*n*), линейно-логарифмическое O(*n* log *n*) и квадратичное время O(*n*^2)) и постараемся понять, как они влияют на эффективность алгоритма. Мы также познакомимся с пространственной сложностью — еще одним важным фактором, влияющим на оценку производительности алгоритма. Так что продолжайте читать, ведь вас ждет много интересного!

3.2. Пространственная сложность

Обсуждая алгоритмы, мы учитываем два наиболее важных фактора: временную и пространственную сложности. Временная сложность определяет количество времени, необходимого алгоритму для выполнения, тогда как пространственная учитывает объем памяти, потребляемой алгоритмом от начала до конца обработки данных. Проще говоря, пространственная сложность — показатель объема памяти, необходимого алгоритму или операции для эффективной работы.

Иметь представление о пространственной сложности важно, особенно в ситуациях, когда объем доступной памяти ограничен. При работе с большими наборами данных или ресурсоемкими приложениями объем памяти, необходимый для запуска алгоритма, может стать серьезной проблемой. Понимая пространственную сложность алгоритма, вы сможете оптимизировать его производительность и обеспечить эффективную работу.

Пространственную сложность, как и временную, можно выразить с помощью нотации «O большое». Она позволяет описать верхнюю границу объема памяти, которая требуется алгоритму по мере увеличения размера входных данных.

Чтобы лучше понять эту идею, рассмотрим пример.

Алгоритм 1. Сумма элементов массива

```
def sum_array(numbers):
    total = 0
    for number in numbers:
        total += number
    return total
```

Этот алгоритм вычисляет сумму всех чисел в массиве. Независимо от размера массива, нам нужно постоянное количество памяти: для одной переменной `total` и для одной переменной цикла. Эта пространственная сложность считается `O(1)` или постоянным объемом памяти.

Алгоритм 2. Создание массива накопленных сумм

```
def cumulative_sum_array(numbers):
    cumulative_sum = [0] * len(numbers)
    cumulative_sum[0] = numbers[0]
    for i in range(1, len(numbers)):
        cumulative_sum[i] = cumulative_sum[i-1] + numbers[i]
    return cumulative_sum
```

Второй алгоритм создает новый массив, в котором будет храниться накопленная сумма для каждого индекса. По мере увеличения размера входного массива растет и размер массива `cumulative_sum`. Пространственная сложность здесь линейная и равна $O(n)$, где n — размер входного массива.

Имея представление о пространственной сложности, можно оценить эффективность алгоритма, помимо его временной сложности. В некоторых ситуациях приходится искать компромисс между временной и пространственной сложностями в зависимости от требований приложения. Например, в системах с ограниченным объемом памяти, возможно, придется выбрать алгоритм с большей временной сложностью, но с меньшей пространственной.

Кроме того, важно знать, что пространственная сложность относится и к вспомогательному пространству, и к пространству, занимаемому входными данными. Вспомогательным является дополнительное или временное пространство, используемое алгоритмом во время выполнения. Пространство же, занимаемое входными данными, необходимо для хранения входных переменных. При анализе пространственной сложности мы обычно сосредоточиваемся на вспомогательном пространстве.

Вдобавок нужно понимать, что временная и пространственная сложности часто находятся в компромиссном соотношении, известном как компромисс между пространством и временем.

Данный компромисс в информатике относится к случаям, когда алгоритм использует больше памяти (более высокая пространственная сложность) для уменьшения времени работы (более низкая временная сложность), и наоборот. Рассмотрим пример, иллюстрирующий этот компромисс.

3.2.1. Кэширование/мемоизация

Кэширование, или мемоизация, — невероятно полезный метод оптимизации производительности алгоритма. По сути, он заключается в хранении результатов дорогостоящих вызовов функций, чтобы алгоритм мог повторно использовать их при повторной обработке тех же входных данных, а не вычислять их заново.

Этот метод позволяет существенно уменьшить количество вычислений, производимых алгоритмом, что, в свою очередь, может значительно повысить его скорость и эффективность. Метод особенно часто используется для решения задач динамического программирования, где с его помощью можно заметно сокращать время и потребляемые ресурсы, необходимые для решения сложных вычислительных задач.

Фактически во многих передовых алгоритмах и структурах данных в информатике кэширование и мемоизация используются для достижения оптимальной производительности. Поэтому владеть данным методом важно любому программисту или информатику, стремящемуся оптимизировать свой код.

Например, рассмотрим алгоритм вычисления n-го числа в последовательности Фибоначчи, где каждое число является суммой двух предыдущих. Алгоритм без мемоизации имеет экспоненциальную временную сложность из-за необходимости выполнять повторяющиеся вычисления.

```
dev fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Но, применив мемоизацию, можно организовать хранение ранее вычисленных результатов в массиве и повторно использовать их, уменьшив временную сложность до линейной. Однако при этом хранение результатов требует дополнительного пространства, что увеличивает пространственную сложность.

```
dev fib(n, memo = {}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    else:
        memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

В этом примере демонстрируется компромисс между пространством и временем. Используя больше памяти, мы выигрываем во времени.

Чтобы писать эффективный код, важно понимать, как пространственная сложность связана с временной. В зависимости от контекста и конкретных ограничений системы или приложения (например, доступной памяти или требований к скорости) вам, возможно, придется тщательно оценить компромисс между временем и пространством, чтобы выбрать или разработать наиболее подходящий алгоритм.

При анализе алгоритмов не всегда нужно стремиться к лучшей временной или пространственной сложности. Ограничения реального мира часто диктуют необходимость идти на компромиссы. В одних случаях можно использовать немного больше памяти, чтобы добиться значительного сокращения времени, особенно при работе с крупными наборами данных. В других из-за аппаратных ограничений необходимо оптимизировать потребление памяти, даже если это означает, что выполнение алгоритма займет немного больше времени.

Кроме того, компромисс между пространством и временем не всегда означает, что экономия времени достигается за счет пропорционального увеличения потребления памяти или наоборот. В одних случаях можно значительно снизить временную сложность, слегка увеличив пространственную, тогда как в других даже заметное увеличение пространственной сложности может привести лишь к небольшой экономии времени. Конкретные результаты зависят от характеристик алгоритма и задачи, которую он решает.

Наконец, помните, что это не единственные факторы, влияющие на выбор алгоритма. Другие аспекты, такие как используемый язык программирования, возможности оборудования, навыки разработчиков и т. д., тоже могут играть важную роль при принятии решения о выборе алгоритма.

Далее мы рассмотрим дополнительные понятия, связанные с эффективностью алгоритма, такие как нотация «О большое», сценарии худшего и среднего случая и многое другое. Кроме того, мы попрактикуемся в определении временной и пространственной сложностей различных алгоритмов и исследуем стратегии оптимизации этих факторов. Приготовьтесь к новым алгоритмическим приключениям!

Помните: наша книга призвана помочь вам понять алгоритмы. Это руководство, ресурс и набор инструментов — все в одном. Вам необязательно

иметь полное понимание каждой идеи в тот момент, когда вы впервые с ней столкнетесь. Мир алгоритмов огромен и сложен. Но в то же время исследовать его очень интересно и чрезвычайно полезно. Продолжайте читать и практиковаться. И самое главное — получайте удовольствие!

3.3. Введение в нотацию «О большое»

Нотация «О большое» — одно из самых важных понятий в информатике. Она помогает описать, как эффективность алгоритма меняется по мере увеличения объема входных данных.

Нотация позволяет нам получить общее представление об алгоритме и выявить верхнюю границу временной или пространственной сложностей в худшем сценарии. Зная эту границу, мы можем понять масштабируемость алгоритма и то, как он будет вести себя при увеличении объема входных данных.

С помощью нотации «О большое» можно предсказать, сколько времени и памяти потребует алгоритм, и сравнить его с другими, чтобы определить лучший для конкретной задачи. Кроме того, эта нотация не зависит от языка программирования и может использоваться для описания эффективности алгоритма на любой платформе.

Таким образом, нотация «О большое» помогает информатикам и инженерам анализировать и оптимизировать алгоритмы, чтобы их можно было использовать в различных приложениях.

Рассмотрим следующий пример:

```
def find_max(numbers):
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num
```

Эта простая функция на Python находит максимальное число в списке. Если в нем хранится n чисел, то в худшем случае функции потребуется выполнить n сравнений, чтобы найти максимальное. Соответственно, мы говорим, что эта функция имеет временную сложность $O(n)$.

3.3.1. Распространенные типы временной сложности

Далее перечислены наиболее распространенные типы временной сложности, с которыми вы можете столкнуться. Типы приведены в порядке убывания эффективности.

- O(1) — постоянная временная сложность, которая не меняется при изменении объема входных данных. Алгоритм, имеющий постоянную временную сложность, может выполняться за фиксированное время, независимо от того, насколько велик или мал объем входных данных. Часто в качестве примера приводят алгоритм доступа к элементу массива по его индексу. Это связано с тем, что индекс элемента массива напрямую определяет физическое местоположение элемента в памяти, следовательно, к нему можно получить доступ, обойдясь без каких-либо дополнительных вычислений или операций. Усвоив понятие постоянной временной сложности, программисты могут разрабатывать более эффективные алгоритмы и оптимизировать свой код в целях повышения производительности.
- O($\log n$) — логарифмическая временная сложность. Количество операций, выполняемых алгоритмом для решения задачи, растет в логарифмической прогрессии по мере роста объема входных данных. Другими словами, при увеличении объема входных данных количество операций, необходимых для решения задачи, будет увеличиваться не пропорционально, а медленнее. Одним из примеров алгоритма с логарифмической временной сложностью может служить алгоритм двоичного поиска, который может быстро найти целевое значение в упорядоченном списке, многократно деля интервал поиска пополам, пока искомое значение не будет найдено. Однако важно отметить следующее: логарифмическая временная сложность обычно считается эффективной, но не всегда является лучшим вариантом для конкретной задачи и ограничений.
- O(n) — линейная временная сложность. Количество операций растет прямо пропорционально росту объема входных данных. То есть при увеличении объема входных данных время, необходимое для выполнения функции, будет увеличиваться пропорционально, а не экспоненциально. Важно отметить, что алгоритмы с линейным временем не обязательно являются медленными. Скорее их можно охарактеризовать так: их производительность предсказуема и стабильна, поэтому они идеально подходят для ситуаций, когда объем входных данных заранее неизвестен и может значительно меняться.

В нашем примере функция `find_max` проверяет каждый элемент входного списка один раз, что делает ее алгоритмом с линейной временной сложностью $O(n)$. Однако существует множество других алгоритмов, имеющих эту сложность, например поиск определенного элемента в неотсортированном списке или вычисление суммы списка чисел. В этих случаях время, необходимое для выполнения функции, будет увеличиваться линейно по мере роста объема входных данных, но алгоритм останется предсказуемым и последовательным.

В целом алгоритмы с линейной временной сложностью являются важной концепцией в информатике и играют решающую роль во многих приложениях, от анализа данных и машинного обучения до веб-разработки и разработки программного обеспечения. Понимая суть линейной временной сложности, разработчики могут создавать более эффективные алгоритмы, которые могут обрабатывать разные объемы входных данных в широком диапазоне и обеспечивать оптимальную производительность.

- $O(n \log n)$ – линейно-логарифмическая временная сложность. Количество операций растет линейно и логарифмически по мере увеличения объема входных данных. Такие алгоритмы работают значительно лучше, чем алгоритмы с линейной временной сложностью. Благодаря этому линейно-логарифмическая временная сложность идеально подходит для алгоритмов сортировки, поскольку позволяет эффективно сортировать большие наборы данных. QuickSort и MergeSort – два примера алгоритмов сортировки, имеющих такую временную сложность, что позволяет им сортировать большие наборы данных за относительно короткий промежуток времени. Линейно-логарифмическую временную сложность имеют и другие алгоритмы, такие как сжатие данных и поиск, которые способны эффективно обрабатывать большие объемы данных. В целом линейно-логарифмическая временная сложность позволяет оптимизировать производительность алгоритмов в широком спектре приложений.
- $O(n\hat{}2)$ – квадратичная временная сложность. Количество операций, необходимых для завершения алгоритма, растет экспоненциально по мере увеличения объема входных данных. Алгоритмы с вложенными циклами, такие как пузырьковая сортировка или сортировка вставкой, часто классифицируются как алгоритмы с квадратичной временной сложностью. По мере увеличения объема входных данных количество операций, выполняемых такими алгоритмами, увеличивается в геометрической прогрессии, вследствие чего они менее эффективны по части обработки

больших наборов данных. При разработке и реализации алгоритмов важно осознавать их сложность, поскольку выбор алгоритма с более высокой временной сложностью может привести к увеличению времени обработки и снижению общей производительности.

O(2^n) или O(n!) – экспоненциальная или факториальная временная сложность. Количество операций растет экспоненциально или факториально по мере увеличения объема входных данных, что может привести к чрезвычайно низкой производительности при обработке больших наборов входных данных. Алгоритмы с такой временной сложностью могут быть очень медленными и часто встречаются в сложных вычислительных задачах, таких как задача коммивояжера.

Таким образом, временная сложность алгоритма – это показатель, отображающий, как количество операций, необходимых для решения задачи, растет по мере увеличения объема входных данных. Оптимизировать алгоритмы, имеющие временную сложность O(2^n) или O(n!), может быть чрезвычайно трудно, и для этого часто требуются творческие и инновационные подходы.

Следует отметить, что в нотации «О большое» мы сохраняем только член высшего порядка и отбрасываем константы. Например, если алгоритм занимает 2n^2 + 100n + 1000 шагов, то мы говорим, что его временная сложность равна O(n^2).

Нотация «О большое» – мощный инструмент анализа эффективности алгоритмов. Она абстрагирует детали и предоставляет простой способ сравнения разных алгоритмов. Однако это теоретическая модель, и фактическая производительность алгоритма зависит от множества других факторов, таких как аппаратное обеспечение, локальность данных и даже особенности входных данных. Поэтому, хотя нотация «О большое» и имеет значение, она не единственный фактор, который следует учитывать при оценке производительности алгоритма.

Нотация «О большое» дает верхнюю границу временной или пространственной сложности, но представляет худший сценарий. Он не всегда может быть типичным случаем при работе алгоритма. Например, временная сложность быстрой сортировки в худшем случае равна O(n^2), а в среднем – O($n \log n$), поэтому ее часто выбирают среди других алгоритмов сортировки, несмотря на высокую сложность в худшем случае.

Более того, два алгоритма могут иметь одинаковую временную сложность, однако на практике один из них может постоянно работать быстрее другого. Это связано с тем, что нотация «О большое» абстрагирует константы

и члены более низкого порядка, которые тоже могут влиять на общее время выполнения, особенно при обработке небольших объемов входных данных.

Предположим, что у нас есть два алгоритма: один выполняет $2n$ шагов, а второй — $500n$ шагов. Оба алгоритма имеют временную сложность $O(n)$, но при меньших значениях n первый алгоритм будет работать значительно быстрее. Однако по мере роста n разница между двумя алгоритмами будет нивелироваться в относительном выражении.

Другими словами, чтобы решить, какой алгоритм является «лучшим» в практическом смысле, не всегда достаточно одной только нотации «O большое». Она — лишь часть головоломки, инструмент, позволяющий размышлять о том, как масштабируется алгоритм. Однако на реальную производительность могут влиять и другие факторы, такие как константы, архитектура и конкретные данные.

Тем не менее любой инженер-программист или информатик должен разбираться в нотации «O большое». Она дает возможность представить, как масштабируются алгоритмы, и позволяет сделать осознанный выбор при проектировании и реализации программного обеспечения. Помните: универсального алгоритма не существует. Лучший выбор зависит от конкретных требований и ограничений решаемой задачи.

3.3.2. Асимптотический анализ

Асимптотический анализ — это метод описания предельного поведения, составляющий основу теории сложности алгоритмов. Термин «асимптотический» означает сколь угодно близкое приближение к значению или кривой (то есть к какому-то пределу). В информатике таким предельным значением обычно является объем входных данных, стремящийся к бесконечности.

В этом контексте мы обсуждали нотацию «O большое», описывающую верхнюю границу временной сложности алгоритма. Другое ее название — асимптотическая верхняя граница. Однако для того чтобы получить полную картину производительности алгоритма, нужно рассмотреть:

- нотацию «Омега (Ω) большая» — она дает асимптотическую нижнюю границу или лучший сценарий;
- нотацию «Тета (Θ) большая» — она используется, когда верхняя и нижняя границы алгоритма совпадают, вследствие чего сохраняется жесткая граница сложности.

Подводя итог, можно сказать, что при анализе алгоритмов важно учитывать не только лучший сценарий (нижняя граница), но и худший (верхняя граница), а также ожидаемый случай (который может находиться где-то посередине). Нотация «О большое» — лишь один из доступных вам инструментов, хотя и очень важный, позволяющий проводить такого рода анализ.

Имея более глубокое понимание теоретической основы анализа алгоритмов, вы теперь лучше подготовлены к изучению конкретных алгоритмов и их сложностей. Читая книгу, помните об этих фундаментальных понятиях — их можно сравнить с линзами, которые помогают оценить каждый алгоритм. Но не забывайте, что теоретический анализ — лишь одна сторона медали. Эмпирический анализ (то есть анализ фактического времени выполнения) реальных наборов данных тоже имеет решающее значение при оценке практической эффективности алгоритма.

На этом мы завершаем наше исследование временной сложности и знакомство с нотацией «О большое». В следующих главах мы подробнее поговорим о пространственной сложности и искусстве балансирования временем и пространством в алгоритмическом проектировании. Оставайтесь с нами!

3.4. Практические задачи

Подкрепим изучение теории некоторыми задачами. Они призваны помочь вам задуматься о том, как работают алгоритмы, и попрактиковаться в анализе их эффективности с точки зрения временной и пространственной сложностей. Для каждой задачи попробуйте написать псевдокод, определить временную и пространственную сложности и объяснить свои рассуждения.

Задача 1. Линейный поиск

- Реализуйте функцию `linear_search(list, item)`, которая принимает список и элемент и возвращает индекс элемента, если он найден в списке, и `-1` в противном случае.
- *Пример входных данных:* `linear_search([1, 3, 5, 7, 9], 5)`.
- *Пример выходных данных:* `2`.
- Каковы временная и пространственная сложности вашей реализации?

Решение. Эту задачу можно решить, перебирая заданный список и сравнивая каждый его элемент с целевым значением.

```
def linear_search(list, item):
    for i in range(len(list)):
        if list[i] == item:
            return i
    return -1
```

Временная сложность: O(n). В худшем случае потребуется просмотреть все элементы списка.

Пространственная сложность: O(1). Функция не создает никаких новых структур данных, размер которых растет по мере увеличения объема входных данных.

Задача 2. Сумма элементов

- Напишите функцию `sum_elements(list)`, которая принимает список чисел и возвращает их сумму.
- *Пример входных данных:* `sum_elements([1, 2, 3, 4, 5])`.
- *Пример выходных данных:* `15`.
- Каковы временная и пространственная сложности вашей реализации?

Решение

Эту задачу можно решить, перебирая заданный список и прибавляя каждый элемент к промежуточной сумме.

```
def sum_elements(list):
    total = 0
    for item in list:
        total += item
    return total
```

Временная сложность: O(n). Нам нужно просмотреть все элементы списка.

Пространственная сложность: O(1). Функция не создает никаких новых структур данных, размер которых растет по мере увеличения объема входных данных.

Задача 3. Поиск дубликатов

- Реализуйте функцию `find_duplicate(list)`, которая принимает список и возвращает первый найденный повторяющийся элемент. Если дубликатов нет, то она должна вернуть `None`.
- *Пример входных данных:* `find_duplicate([1, 2, 3, 4, 2, 5])`.
- *Пример выходных данных:* `2`.
- Каковы временная и пространственная сложности вашей реализации?

Решение

Для выявления дубликатов можно использовать множество. Столкнувшись с элементом, который во множестве уже есть, мы можем с уверенностью сказать, что это дубликат.

```
def find_duplicate(list)
    seen = set()
    for item in list:
        if item in seen:
            return item
        seen.add(item)
    return None
```

Временная сложность: $O(n)$. Нам нужно просмотреть все элементы списка.

Пространственная сложность: $O(n)$. В худшем случае нам придется сохранить во множестве все элементы из входного списка.

Задача 4. Пузырьковая сортировка

- Напишите функцию `bubble_sort(list)`, которая сортирует список чисел, используя алгоритм пузырьковой сортировки.
- *Пример входных данных:* `bubble_sort([5, 1, 4, 2, 8])`.
- *Пример выходных данных:* `[1, 2, 4, 5, 8]`.
- Подумайте о том, каковы временная и пространственная сложности вашей реализации.

Решение

Мы можем реализовать алгоритм пузырьковой сортировки, который неоднократно проходит по списку, сравнивает соседние элементы и меняет их местами, если они стоят в неправильном порядке.

```
def bubble_sort(list):
    n = len(list)
    for i in range(n):
        for j in range(0, n - i - 1):
            if list[j] > list[j + 1]:
                list[j], list[j + 1] = list[j + 1], list[j]
    return list
```

Временная сложность: O(n^2). Каждый элемент нужно сравнить с каждым другим элементом.

Пространственная сложность: O(1). Функция не создает никаких новых структур данных, размер которых растет по мере увеличения объема входных данных.

Решая задачи, думайте не только о правильности решения, но и о его эффективности. Постарайтесь оптимизировать свои решения как по времени, так и по потребляемой памяти, где это возможно. Учитывайте пограничные случаи: например, что произойдет, если входной список пуст или все элементы в списке одинаковы?

Решив эти задачи, вы будете лучше разбираться в практическом анализе эффективности алгоритмов и подготовитесь к встрече с более сложными задачами и алгоритмами!

Резюме

Эффективность алгоритмов — основа, на которой строится вся информатика. Умение измерить эффективность алгоритма с точки зрения временной и пространственной сложностей крайне важно для любого начинающего программиста или информатика. В этой главе мы углубились в данную тему и исследовали ее теоретические и практические аспекты.

Сначала мы обсудили временную сложность. Мы увидели, что это показатель того, как количество времени, необходимого алгоритму для выполнения, увеличивается в зависимости от роста объема входных данных. Имея глубокое понимание временной сложности, мы можем оценивать и сравнивать алгоритмы по их производительности и выбирать наиболее эффективные.

Мы подкрепили обсуждение подробным исследованием линейного и бинарного поиска, подчеркнув, насколько важно понимать, как временная сложность меняется при увеличении объема входных данных.

Затем мы обратили внимание на пространственную сложность — показатель объема памяти, необходимого для выполнения алгоритма. Пространственная сложность так же важна, как и временная. Иногда оптимизация алгоритма, призванная уменьшить пространственную сложность, может привести к увеличению временной, и наоборот. Научиться находить хрупкий баланс между ними — важный навык для любого информатика.

После этого мы познакомились с нотацией «О большое» — базовым инструментом, который используется для описания эффективности алгоритма. Мы узнали, что эта нотация представляет верхнюю границу временной сложности и позволяет рассуждать о сценарии наихудшей производительности алгоритма. От того, насколько хорошо вы понимаете нотацию «О большое», зависит оценка масштабируемости алгоритмов, особенно при работе с большими объемами входных данных.

Продолжая знакомиться с нотацией «О большое», мы кратко обсудили родственные нотации «Омега большая» и «Тета большая», представляющие нижнюю границу (наилучший сценарий) и жесткую границу (обе границы, лучшая и худшая, совпадают) соответственно.

Благодаря практическим задачам, представленным в этой главе, вы смогли применить новые теоретические знания на практике и еще более глубоко вникнуть в суть рассмотренных концепций. Работа над задачами помогла разобраться с временной и пространственной сложностями и увидеть, как эти понятия влияют на выбор решений задач.

В заключение отметим, что умение анализировать временную и пространственную сложности алгоритма и выражать этот анализ с помощью нотации «О большое» является основополагающим в информатике. По мере перехода к более сложным алгоритмам и структурам данных помните об этих концепциях, поскольку они формируют фундамент, на котором строится все остальное. Зачастую цель состоит не в том, чтобы просто найти решение, а в том, чтобы найти наиболее эффективное. Если вы поняли, как оценивается эффективность алгоритмов, то обрели способность принимать обоснованные решения о выборе алгоритмов в различных ситуациях, что позволяет вам писать более эффективный и масштабируемый код.

Глава 4
ОСНОВНЫЕ ТИПЫ АЛГОРИТМОВ

В мире алгоритмов существует множество вариантов выбора. Каждый тип алгоритмов обладает уникальными качествами, характеристиками и областью применения, что делает их незаменимыми инструментами в арсенале программиста.

С одной стороны, алгоритмы «разделяй и властвуй» известны своей способностью разбивать сложные задачи на более мелкие и управляемые подзадачи. С другой стороны, жадные алгоритмы фокусируются на локально оптимальном выборе на каждом этапе, стремясь найти глобальный оптимум.

Алгоритмы динамического программирования нужны для того, чтобы решать задачи путем деления их на более мелкие подзадачи и сохранять результаты последних, таким образом избегая избыточных вычислений. Наконец, алгоритмы прямого перебора — это самые простые алгоритмы, которые пробуют все возможные решения и выбирают лучшее.

Эти фундаментальные и широко используемые типы алгоритмов лежат в основе многих сложных алгоритмов и структур данных, применяемых в информатике, что делает их неотъемлемой частью базы знаний программиста.

4.1. Алгоритмы «разделяй и властвуй»

Первым мы рассмотрим алгоритмы типа *«разделяй и властвуй»*. Эта стратегия широко используется при решении задач и выглядит так: задачу разбивают на более мелкие подзадачи, решают их независимо друг от друга, а затем объединяют решения для того, чтобы решить исходную задачу. Красота этого подхода заключается в его рекурсивном характере, когда каждая подзадача разделяется до тех пор, пока не станет достаточно простой для того, чтобы ее можно было решить напрямую.

Другим примером алгоритма «разделяй и властвуй» может служить алгоритм сортировки слиянием, который используется для сортировки больших наборов данных. Алгоритм делит набор данных на более мелкие части,

сортирует их независимо, а затем объединяет отсортированные части с целью получить окончательный результат. Этот подход особенно эффективен при работе с большими наборами данных, поскольку позволяет сортировать их за более короткое время.

Важно отметить, что с помощью алгоритмов «разделяй и властвуй» можно решать разные задачи: начиная с простой сортировки и заканчивая сложными математическими вычислениями. Алгоритм двоичного поиска, который служит классическим примером подхода «разделяй и властвуй», широко используется в информатике для поиска в отсортированных наборах данных. Деля пространство поиска пополам на каждом шаге, алгоритм двоичного поиска способен быстро отыскать целевое значение.

Таким образом, алгоритмы «разделяй и властвуй» представляют собой мощную стратегию, которая может применяться для решения самых разных задач. Рекурсивная природа этой стратегии позволяет эффективно разбивать сложные задачи на более мелкие и управляемые подзадачи, что делает ее важным инструментом в арсенале информатиков и математиков.

Рассмотрим псевдокод алгоритма двоичного поиска:

```
function binary_search(list, item):
    low = 0
    high = length of list - 1

    while low <= high:
        mid = (low + high) / 2
        guess = list[mid]

        if guess is item:
            return mid
        if guess > item:
            high = mid - 1
        else:
            low = mid + 1

    return None
```

В этом псевдокоде задача поиска элемента в отсортированном списке разбивается на более мелкие подзадачи (поиск в нижней или верхней половине списка). Этот процесс продолжается до тех пор, пока элемент не будет найден или пока пространство поиска не станет пустым.

Другим широко известным алгоритмом «разделяй и властвуй» является алгоритм QuickSort (быстрая сортировка). Он выбирает «опорный» элемент в массиве и делит другие элементы массива на два подмассива в зависимости

от того, меньше они или больше опорного элемента. Затем алгоритм рекурсивно сортирует подмассивы.

```
function quicksort(array):
    if length of array < 2:
        return array
    else:
        pivot = array[0]
        less = [ i for i in array[1:] if i<= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

В этом псевдокоде функция `quicksort` сначала проверяет, содержит ли входной массив меньше двух элементов. Если да, это означает, что массив уже отсортирован и его можно просто вернуть. Если массив имеет два или несколько элементов, то функция выбирает первый элемент в качестве опорного. Затем она делит остальную часть массива на два подмассива: один — с элементами меньше опорного, а второй — с элементами больше опорного. Получившиеся массивы сортируются рекурсивно и объединяются с опорным элементом, чтобы можно было получить отсортированный массив.

Эти примеры иллюстрируют возможности стратегии «разделяй и властвуй»: она может значительно снизить временную сложность алгоритмов, особенно при обработке больших объемов входных данных. Далее мы углубим наше понимание алгоритмов «разделяй и властвуй», рассмотрев несколько практических примеров и выполнив ряд упражнений.

Обсудим некоторые важные свойства и следствия стратегии «разделяй и властвуй».

Рекурсивная природа

Как видно из предыдущих примеров, алгоритмы «разделяй и властвуй» естественным образом реализуются в форме рекурсивных функций. Этот рекурсивный характер позволяет алгоритмам хорошо приспосабливаться к размерам задач, разбивая их на более мелкие, решая каждую подзадачу независимо, а затем объединяя решения, чтобы получить общее решение исходной задачи.

Как вы уже знаете, этот процесс можно повторять рекурсивно до тех пор, пока размер задачи не станет достаточно малым, чтобы ее можно было решить напрямую. Рекурсивные функции каждый раз вызывают себя с разными аргументами, что позволяет им обрабатывать каждую подзадачу независимо.

В результате получаются эффективные алгоритмы, которые могут справиться с задачами большего размера. Таким образом, рекурсивная природа алгоритмов «разделяй и властвуй» является важным фактором их успеха и масштабируемости.

Эффективность

Алгоритмы «разделяй и властвуй» часто оказываются более действенными, чем простые итеративные решения, поскольку разбивают задачу на более мелкие части, благодаря чему могут эффективнее использовать вычислительные ресурсы. Используя параллелизм и распределяя рабочую нагрузку между несколькими процессорами, алгоритмы «разделяй и властвуй» могут значительно ускорить вычисления. Более того, они решают более мелкие подзадачи по отдельности, что часто приводит к снижению временной сложности.

Эта стратегия делает алгоритмы «разделяй и властвуй» особенно полезными для решения задач с большим количеством подзадач, поскольку помогает уменьшить количество необходимых вычислений. Например, двоичный поиск имеет временную сложность $O(\log n)$, тогда как простой линейный поиск — $O(n)$.

При меньшей временной сложности алгоритмы «разделяй и властвуй» можно использовать для решения задач, требующих большого объема вычислений, таких как обработка изображений, машинное обучение и научное моделирование.

Потребление памяти

Алгоритмы «разделяй и властвуй» известны своей эффективностью в решении сложных задач, но не всегда оказываются лучшим вариантом, особенно при использовании в средах с ограниченным объемом памяти. Причина в том, что им часто требуется дополнительное пространство на стеке, чтобы выполнять рекурсивные вызовы, а это может увеличить потребление памяти. Однако есть стратегии, помогающие решить эту проблему.

Например, для хранения ранее вычисленных значений и уменьшения потребности в дополнительной памяти можно использовать мемоизацию. Кроме того, некоторые алгоритмы «разделяй и властвуй», такие как алгоритм Штрассена для умножения матриц, прошли оптимизацию, призванную уменьшить потребление памяти. Но, несмотря на эти соображения, при выборе подхода к решению конкретной задачи важно тщательно оценить компромисс между потреблением памяти и эффективностью алгоритма.

Параллелизм

Одним из значительных преимуществ алгоритмов «разделяй и властвуй» является их способность распараллеливаться, то есть решать различные подзадачи независимо друг от друга на нескольких процессорах.

Такое распараллеливание может существенно увеличить эффективность, особенно в крупномасштабных задачах, где решение подзадач требует значительных вычислительных ресурсов. Параллелизм может сократить общее время, необходимое для решения задачи, что является важным фактором в случаях, когда время имеет большое значение.

Благодаря этому свойству алгоритмы «разделяй и властвуй» отлично подходят для высокопроизводительных вычислительных приложений, где важны точность и скорость.

Алгоритмы «разделяй и властвуй» хороши тем, что они простые и масштабируемые. Они позволяют подходить к решению сложных задач системно, что делает их важным инструментом в арсенале каждого информатика.

К настоящему моменту мы заложили прочную основу, благодаря которой сможем понять алгоритмы «разделяй и властвуй», изучили их базовую стратегию, перечислили присущие им свойства и увидели их в действии на примерах двоичного поиска и быстрой сортировки.

Для полноты картины кратко перечислим некоторые другие известные алгоритмы «разделяй и властвуй», которые вы, возможно, захотите изучить самостоятельно.

Сортировка слиянием

Это алгоритм сортировки, следующий парадигме «разделяй и властвуй», которая предполагает разделение массива на более мелкие подмассивы, их сортировку по отдельности и последующее объединение. Сортировка слиянием делит массив на две половины, сортирует их по отдельности, а затем объединяет.

Этот процесс рекурсивно выполняется для двух половин до тех пор, пока не будет достигнут базовый случай, то есть когда в подмассиве останется только один элемент. Производительность сортировки слиянием обычно

составляет $O(n \log n)$, что выше, чем у большинства других популярных алгоритмов сортировки. Зато это очень стабильный алгоритм сортировки, и он сохраняет относительный порядок равных элементов в массиве.

Сортировка слиянием широко используется в различных вычислительных приложениях, включая сетевую маршрутизацию и сжатие файлов.

Алгоритм Штрассена

Этот широко известный алгоритм умножения матриц, особенно эффективный для больших матриц, впервые был предложен Фолькером Штрассеном в 1969 году. Алгоритм делит большую матрицу на меньшие и выполняет необходимые операции. Такой подход может уменьшить количество вычислений, необходимых для умножения двух матриц, по сравнению с традиционным методом.

Алгоритм был детально изучен и нашел практическое применение в таких областях, как информатика, инженерное дело и физика. Благодаря своей способности обрабатывать большие матрицы он стал часто использоваться во многих приложениях.

Однако важно отметить, что этот алгоритм не всегда оказывается наиболее эффективным способом умножения матриц, особенно если они небольшого размера. Поэтому важно тщательно учитывать размер матриц, прежде чем использовать данный алгоритм.

Алгоритм Карацубы

Это один из наиболее эффективных алгоритмов умножения. Используя подход «разделяй и властвуй», он разбивает задачу умножения на более мелкие и более управляемые части. Этот подход особенно эффективен при перемножении больших чисел, где традиционные методы умножения могут оказаться медленными и громоздкими.

Разбивая задачу на более мелкие части, алгоритм Карацубы способен ускорить процесс умножения и повысить общую эффективность вычислений. Этот алгоритм активно применяется в таких областях, как криптография, информатика и инженерия, где важна способность быстро и точно выполнять сложные вычисления.

В целом алгоритм Карацубы — это мощный инструмент, который произвел революцию в подходах к решению задач умножения, и в ближайшие годы он наверняка будет играть важную роль во многих областях.

Ханойские башни

Это классический рекурсивный алгоритм. Он использует метод «разделяй и властвуй» для решения задачи за минимальное количество ходов.

Ханойские башни — это математическая головоломка, служащая классическим примером рекурсии с тех пор, как она была впервые представлена Эдуардом Лукасом в 1883 году. Головоломка состоит из трех стержней и нескольких дисков разного размера, которые можно надевать на любой стержень. Решение головоломки начинается с того, что все диски надеваются на один стержень в порядке уменьшения размеров, снизу вверх, и в результате образуется коническая форма.

Цель головоломки — переместить все диски на другой стержень, соблюдая следующие простые правила.

1. За один ход можно переместить только один диск.
2. Каждый ход состоит в том, чтобы снять верхний диск с одного из стержней и надеть его на другой.
3. Никакой диск нельзя надеть на стержень, если под ним окажется диск меньшего размера.

Говорят, что эту головоломку придумал французский математик, но ее происхождение до сих пор остается предметом споров. В любом случае она позволяет попрактиковаться в решении задач и развить критическое мышление. Используя метод «разделяй и властвуй», головоломку можно решить за минимальное количество ходов. Головоломка использовалась в информатике, программировании и анализе алгоритмов. Она существует и в виде популярной игрушки, которую можно найти во многих магазинах игрушек по всему миру.

Пара ближайших точек

Эта задача широко распространена в вычислительной геометрии и заключается в поиске двух точек из заданного набора на плоскости x–y, которые находятся ближе всего друг к другу. Задачу можно решить за время O(n^2),

что не очень эффективно для больших наборов данных, однако существует более эффективный подход, основанный на методе «разделяй и властвуй».

Он позволяет решить задачу за время $O(n \log n)$, что будет намного быстрее, и поэтому лучше подходит для обработки больших наборов данных. Метод «разделяй и властвуй» предполагает разделение набора точек на более мелкие подмножества и последующее решение задачи на каждом подмножестве отдельно. Отыскав пары ближайших точек в каждом подмножестве, алгоритм объединяет результаты, чтобы найти ближайшую пару точек во всем множестве.

Для работы с небольшими наборами данных этому методу может потребоваться больше времени, чем методу простого перебора. Но он гораздо лучше масштабируется по мере увеличения объема данных, поэтому больше подходит для решения этой задачи на практике.

Итак, вы увидели, как алгоритмы «разделяй и властвуй» применяются в различных областях: при математических вычислениях, сортировке и поиске данных и решении сложных математических головоломок.

Теперь, получив представление об этих алгоритмах, перейдем к следующему типу: жадным алгоритмам. Наше путешествие продолжается, и, как всегда, ключевым моментом является практика. Я рекомендую вам попробовать написать и выполнить эти алгоритмы самостоятельно, чтобы лучше их понять.

4.2. Жадные алгоритмы

Жадные алгоритмы — это высокоэффективный и интуитивно понятный подход к решению определенных типов задач, особенно связанных с оптимизацией. Основной принцип жадных алгоритмов таков: выбрать наилучший вариант, доступный на каждом шаге, чтобы выработать лучшее общее решение.

Этот подход может пригодиться в ситуациях, когда решение необходимо найти быстро или когда сама задача очень сложна. Благодаря возможности разбить задачу на более мелкие шаги и на каждом из них выбрать лучшее решение жадные алгоритмы могут оказаться наиболее эффективным вариантом.

Кроме того, данный подход можно адаптировать к решению самых разных задач в различных сценариях, что делает его универсальным инструментом.

В целом жадные алгоритмы могут значительно повысить эффективность решения задач и важны для любого человека или организации, стремящихся достичь оптимальных результатов в своей работе.

4.2.1. Что такое жадный алгоритм

В области информатики, и особенно в области алгоритмов, жадным называют алгоритм, который «жадно» выбирает оптимальный результат в каждой точке принятия решения.

По сути, жадный алгоритм предполагает пошаговый подход к решению задачи, когда мы делаем выбор, который выглядит лучше всего в данный конкретный момент, в надежде, что этот выбор в итоге приведет к наилучшему решению задачи. Один из ключевых аспектов жадных алгоритмов — они принимают локально оптимальные решения, которые считаются окончательными и не могут быть отменены или пересмотрены в будущем.

Чтобы лучше понять, как работают жадные алгоритмы, рассмотрим классическую задачу размена монет. Суть задачи такова: нужно найти минимальное количество монет, необходимое для того, чтобы набрать определенную сумму сдачи. Используя для решения этой задачи жадный алгоритм, можно начать с выбора монеты наибольшего номинала, которая меньше или равна оставшейся сумме сдачи.

Этот процесс повторяется до тех пор, пока оставшаяся сумма сдачи не станет равна нулю. Данный подход не всегда приводит к оптимальному решению, но зачастую оказывается относительно быстрым и достаточно эффективным способом решения задачи.

4.2.2. Задача размена монет

Перед вами как программистом, которому поручено создать автомат для выдачи сдачи наименьшим возможным количеством монет, стоит непростая задача. Монеты, с которыми вам предстоит работать, имеют номинал 1, 5, 10 и 25 центов. Для решения задачи можно использовать жадный алгоритм.

Алгоритм можно разбить на следующие этапы.

1. Выбрать монету наибольшего номинала, который меньше оставшейся суммы сдачи.

2. Вычесть номинал монеты из оставшейся сдачи.
3. Повторять процедуру, пока оставшаяся сдача не станет равной нулю.

Следуя этому алгоритму, вы можете быть уверены, что автомат выдает сдачу, используя наименьшее количество монет. Это объясняется тем, что алгоритм отдает приоритет монетам с наибольшим номиналом, что уменьшает общее количество монет в сдаче.

Реализуем этот алгоритм:

```
def greedy_coin_change(coins, amount):
    coins.sort(reverse=True)
    result = []

    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)

    return result

coins = [1, 5, 10, 25]
amount = 63
print(greedy_coin_change(coins, amount)) # Выведет: [25, 25, 10, 1, 1, 1]
```

Функция `gredy_coin_change` использует жадную стратегию для поиска минимального количества монет, составляющих заданную сумму. Она начинается с сортировки номиналов монет в порядке убывания. Затем входит в цикл, перебирающий номиналы. Внутри него, пока оставшаяся сумма больше или равна текущему номиналу монеты, этот номинал вычитается из суммы и добавляется в список результатов. Функция возвращает список номиналов монет, использованных для сдачи.

Этот подход хорошо работает при определенном наборе номиналов монет. В некоторых случаях жадная стратегия не дает оптимального решения (например, если бы у нас были монеты с номиналом 1, 3 и 4 цента и требовалось дать сдачу 6 центов, то лучшим решением было бы дать две монеты с номиналом 3 цента, но жадная стратегия даст одну монету 4 цента и две монеты по 1 центу).

Задача размена монет — классика введения в жадные алгоритмы, но вам предстоит узнать куда больше. В следующих разделах мы обсудим более сложные задачи и рассмотрим случаи, когда жадные стратегии будут либо оптимальными, либо нет. Вы научитесь определять, какие задачи можно решить, используя жадный подход, а какие — нет.

В дополнение к вышесказанному рассмотрим еще несколько примечательных примеров жадных алгоритмов и вариантов их использования.

- **Алгоритм кодирования Хаффмана.** Это широко используемый метод сжатия данных, позволяющий значительно уменьшить размер данных, при этом все необходимые детали сохраняются. Это тип сжатия без потерь, который работает путем создания оптимального кодирования без префиксов.

  Алгоритм является жадным и присваивает более короткие коды наиболее часто встречающимся символам, а более длинные — наименее часто встречающимся символам. Такой подход гарантирует, что часто встречающиеся символы будут представлены меньшим количеством битов, что помогает уменьшить общий размер данных.

  Этот метод особенно эффективен для приложений, где пространство хранения ограничено или когда данные необходимо быстро передавать по сети. Кроме того, кодирование Хаффмана часто используется в сочетании с другими методами сжатия, что позволяет достичь еще большей степени сжатия. Используя кодирование Хаффмана, пользователи могут эффективно уменьшить объем своих данных, не теряя никакой важной информации.

- **Алгоритм минимального остовного дерева Прима.** Этот алгоритм поиска минимального остовного дерева для взвешенного неориентированного графа широко используется в информатике. Процесс начинается с инициализации минимального связующего дерева с одной вершиной.

  Затем алгоритм продолжает добавлять следующие вершины к минимальному остовному дереву. Хитрость в том, что следующая добавляемая вершина должна иметь минимальное ребро, которого еще нет в минимальном остовном дереве.

  Этот метод может пригодиться в случаях, когда граф слишком велик, чтобы его можно было пройти вручную, или когда на перемещение по графу вручную требуется непомерно много времени. Кроме того, данный алгоритм часто используется при разработке компьютерных сетей, где важно найти минимальное остовное дерево и таким образом оптимизировать производительность сети.

- **Алгоритм минимального остовного дерева Краскала.** Альтернативный метод получения минимального остовного дерева из графа — «жадный алгоритм». Этот подход рассматривает каждый узел графа как отдельное дерево и устанавливает связь между узлами, только если она оказывается

наиболее эффективной с точки зрения стоимости. Для этого алгоритм итеративно исследует все возможные ребра графа, выбирая то, которое имеет наименьшую стоимость.

После создания связи алгоритм повторяет процесс, но только с оставшимися узлами и ребрами, которые еще не являются частью дерева. Так продолжается до тех пор, пока все узлы не будут связаны. Этот метод не всегда приводит к минимальному остовному дереву, но является достаточно эффективным и широко используется для получения приближенного решения.

- **Алгоритм Дейкстры для поиска кратчайшего пути.** Алгоритм Дейкстры — широко признанный и высокоэффективный алгоритм поиска, который обычно используется для определения кратчайшего пути между двумя узлами графа. Этот алгоритм применяется множеством способов, выходящих далеко за рамки его традиционного использования для поиска маршрутов, в том числе служит в роли подпрограммы в других графовых алгоритмах.

  Изначально алгоритм разрабатывался для использования на графе с одним источником без отрицательных весов, но впоследствии был адаптирован для работы с различными типами графов и стал важным инструментом в информатике и смежных областях. Благодаря своей универсальности он часто используется при оптимизации сетей, в транспортной логистике и даже робототехнике.

  Более того, многие исследователи в настоящее время изучают инновационные способы улучшения данного алгоритма, закладывая основы создания еще более совершенных и мощных алгоритмов поиска.

- **Задача о рюкзаке.** Согласно задаче о рюкзаке, мы можем дробить предметы, чтобы увеличить общую стоимость содержимого рюкзака. Жадная стратегия, реализуемая этим алгоритмом, предназначена для решения задач оптимизации, когда лучший выбор, совершаемый на каждом шаге, в итоге приводит к оптимальному решению.

Обратите внимание: несмотря на эффективность, жадные алгоритмы не всегда дают оптимальное решение каждой конкретной задачи. Иногда они могут приводить к очень плохим решениям. Поэтому важно понимать суть задачи и применимость алгоритма.

Помните: понять и освоить жадные алгоритмы позволяет практика. Пробуйте применять эту идею к другим задачам — и со временем поймете, когда нужно использовать жадную стратегию, а когда — нет.

В качестве последнего замечания о жадных алгоритмах следует подчеркнуть важность понятий «свойства жадного выбора» и «оптимальная подструктура».

Свойство жадного выбора. Одна из наиболее важных особенностей жадных алгоритмов. Эта ключевая характеристика позволяет алгоритму делать выбор, который выглядит лучшим вариантом в данный момент, и при этом сохранять конечную цель — поиск наилучшего возможного решения всей задачи. По сути, алгоритм является «жадным», поскольку последовательно делает выбор, который, по его мнению, приведет к наилучшему результату на каждом этапе процесса. Поступая таким образом, он пытается найти общее оптимальное решение задачи. Именно благодаря этому свойству жадные алгоритмы могут успешно справляться с работой и быть полезными в широком спектре приложений.

Оптимальная подструктура. Это понятие играет важную роль в разработке алгоритмов. Оптимальное решение конкретной задачи может быть найдено путем объединения оптимальных решений ее подзадач. Это свойство особенно полезно при выборе оптимального алгоритмического подхода к решению задачи, поскольку подчеркивает потенциальную эффективность динамического программирования и жадных алгоритмов. По сути, оптимальная подструктура помогает разбивать сложные задачи на более мелкие и управляемые подзадачи, что позволяет решать их более эффективно.

При разработке жадного алгоритма для новой задачи крайне важно иметь представление об этих двух свойствах. Более того, применимость жадного алгоритма во многом определяется тем, есть ли у задачи эти свойства. Если нет, то жадный алгоритм может не дать оптимального решения.

Кроме того, не забывайте, что жадные алгоритмы не универсальны, даже притом что часто приводят к высокоэффективным решениям. В некоторых случаях лучшие результаты можно получить, используя другие типы алгоритмов, такие как динамическое программирование или принцип «разделяй и властвуй». Вот зачем вы изучаете различные классы алгоритмов — чтобы в вашем арсенале был полный набор инструментов для решения разных задач!

В следующем разделе описывается динамическое программирование, благодаря которому вы еще больше расширите свои возможности по решению задач.

4.3. Алгоритмы динамического программирования

Динамическое программирование (ДП) — это алгоритмический метод разбиения задачи оптимизации на более простые подзадачи. Решение общей задачи зависит от оптимального решения ее подзадач.

Преимущество ДП перед алгоритмами «разделяй и властвуй» заключается в том, что ДП применимо к зависимым подзадачам. Другими словами, когда подзадачи имеют общие подподзадачи, ДП будет работать, а алгоритмы «разделяй и властвуй» — нет.

Алгоритмы ДП решают подзадачи только раз и сохраняют ответ в таблице. Это избавляет от необходимости заново вычислять ответ каждый раз, когда встречается уже решенная подзадача. Так можно сэкономить время и высвободить вычислительные мощности для решения других задач.

ДП — эффективный метод, активно применяющийся в таких областях, как информатика, экономика и инженерия. Разбивая сложные задачи на более мелкие и управляемые подзадачи, ДП позволяет решать задачи, которые в ином случае было бы слишком сложно решить.

Рассмотрим задачу вычисления последовательности чисел Фибоначчи. Вот простое рекурсивное решение:

```
def fibonacci(n):
    if n <= 1:
        return n
    else
        return (fibonacci(n-1) + fibonacci(n-2))
```

Этот алгоритм надежный, но неэффективный. Он выполняет слишком много повторяющихся вычислений. Если нарисовать дерево рекурсии, то можно обнаружить, что одни и те же подзадачи выполняются по несколько раз.

Решить эту проблему можно с помощью динамического программирования, двумя способами: сверху вниз (мемоизация) и снизу вверх.

Решение сверху вниз (мемоизация)

В информатике подход сверху вниз часто используется для решения сложных задач. Он предполагает разбиение большой задачи на более мелкие подзадачи, которые затем решаются одна за другой.

Этот метод особенно полезен при решении задач, имеющих перекрывающиеся подзадачи. Чтобы избежать избыточных вычислений, используется метод мемоизации — формы кэширования, когда результаты дорогостоящих вызовов функций сохраняются и возвращаются в случае необходимости выполнить вычисления для тех же входных данных. Благодаря этому алгоритм может избежать многократного выполнения вычислений при решении одной и той же подзадачи и повысить общую эффективность.

Вот как можно реализовать вычисление чисел Фибоначчи с помощью динамического программирования, если использовать метод сверху вниз:

```
def fibonacci(n, memo):
    if n <= 1:
        return n
    elif n not in memo:
        memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)

    return memo[n]
```

Решение снизу вверх

Это противоположный подход. Вычисления начинаются с самой простой подзадачи и итеративно переходят к все более крупным подзадачам. Задачи предварительно разбиваются на простейшие подзадачи, а затем с помощью итеративного процесса решаются все более крупные подзадачи. Подход часто используется в динамическом программировании, где решение задачи строится на решениях предыдущих подзадач.

Начав с самых мелких подзадач, мы можем постепенно подойти к решению всей задачи и получить оптимальное решение. В отличие от подхода сверху вниз, который последовательно разбивает задачу на более мелкие подзадачи, подход снизу вверх предлагает более систематический и организованный процесс решения, гарантирующий, что ни одна подзадача не будет упущена из виду.

Вернемся к задаче вычисления чисел Фибоначчи и решим ее, используя метод динамического программирования снизу вверх:

```
def fibonacci(n):
    fib = [0. 1] + [0] * (n-1)
    for i in range(2, n+1):
        fib[i] = fib[i-1] + fib[n-2]
    return fib[n]
```

Оба подхода дают один и тот же ответ, но получают его разными способами. Подход сверху вниз обычно легче понять, поскольку он ближе к исходной постановке задачи. Подход снизу вверх часто более эффективен, так как устраняет накладные расходы на рекурсию.

Динамическое программирование — эффективный метод, позволяющий решать многие типы задач за время O(n^2), O(n^3) или даже за линейное время, что в противном случае было бы невозможно. Примерами таких задач служат задача о рюкзаке, задача коммивояжера, умножение цепочки матриц и т. д.

Следует помнить, что динамическое программирование применимо только тогда, когда каждая подзадача дискретна и не зависит от других. Благодаря этой особенности ДП можно использовать для решения задач, в которых существует постоянное состояние, влияющее на решение.

В качестве дополнительного замечания по динамическому программированию стоит упомянуть, что процесс разработки алгоритмов ДП можно разбить на несколько шагов.

1. Охарактеризовать структуру оптимального решения.
2. Определить оптимальность рекурсивного решения с точки зрения более мелких подзадач.
3. Вычислить значение оптимального решения (обычно методом снизу вверх).
4. Построить оптимальное решение задачи на основе вычисленной информации.

Стоит отметить и тот факт, что динамическое программирование в основном используется, когда решение можно повторно выразить через предыдущие результаты. Это делает ДП ценным в случаях, когда количество повторяющихся подзадач в методе прямого перебора возрастает экспоненциально.

Однако динамическое программирование не всегда оказывается наиболее эффективным и практичным методом решения. Его лучше применять к задачам, которые можно разделить на более мелкие, и эти подзадачи «используются повторно» несколько раз.

В следующем разделе мы углубимся в рекурсивные алгоритмы — еще одну распространенную и фундаментальную категорию алгоритмов, используемых в информатике. По мере чтения книги вы получите четкое представление

о различных типах алгоритмов и сможете определять, какой из них лучше всего подходит для той или иной задачи. Этот навык является ключевым в области информатики и разработки программного обеспечения и пригодится вам в любой смежной области.

4.4. Рекурсивные алгоритмы

Рекурсивные алгоритмы — это класс алгоритмов, решающих задачу путем решения меньших экземпляров одной и той же задачи. Другими словами, рекурсивный алгоритм разбивает задачу на более мелкие части до тех пор, пока те не станут достаточно простыми, чтобы их можно было решить напрямую. Эта стратегия основана на принципе «разделяй и властвуй».

Идея рекурсии занимает в программировании одно из центральных мест. Рекурсия позволяет функции вызывать саму себя, создавая итеративный цикл, явная конструкция цикла не используется. Однако важно определить базовый случай или простую подзадачу, которую можно решить без дальнейшего ее деления. Этот базовый случай служит точкой остановки рекурсии, предотвращая бесконечные циклы и гарантируя завершение алгоритма.

На практике рекурсию можно использовать для решения самого широкого круга задач: начиная с поиска в структурах данных и заканчивая сортировкой и даже играми, такими как шахматы. Рекурсия может быть очень эффективным инструментом программирования, но важно использовать ее разумно и понимать имеющиеся ограничения. В частности, рекурсивные алгоритмы могут быть не такими эффективными, как их итеративные аналоги, и их сложнее отлаживать и понимать. Тем не менее рекурсия остается базовым понятием информатики и ценным инструментом в арсенале программиста.

Вот простой пример рекурсивного алгоритма: вычисление факториала числа. Факториал числа n, обозначаемый как $n!$, вычисляется как произведение всех натуральных чисел, меньших или равных n. Его можно определить рекурсивно следующим образом:

```
function factorial(n)
    if n == 0
        return 1
    else
        return n * factorial(n - 1)
    end if
end function
```

Функция `factorial(n)` вызывает себя для вычисления факториала числа `(n - 1)`, и так продолжается до тех пор, пока `n` не станет равным `0`, после чего она вернет `1`. Это базовый случай.

Рекурсия может быть элегантным способом решения задач, имеющих естественную иерархическую или фрактальную структуру, таких как обход деревьев и графов, Ханойские башни или вычисление чисел Фибоначчи.

Однако рекурсию следует использовать разумно. Она может упростить выражение некоторых алгоритмов, но может и привести к таким проблемам, как ошибка переполнения стека, если глубина рекурсии (количество вложенных вызовов функций) окажется слишком большой. Кроме того, при неправильном использовании она может быть неэффективной из-за многократного решения одних и тех же подзадач.

4.4.1. Хвостовая рекурсия

В рекурсивных алгоритмах функция вызывает саму себя, создавая цепочку вызовов, которая может обрабатываться компьютером, пока не будет достигнут базовый случай, останавливающий рекурсию. Когда рекурсивный вызов является последней операцией в функции, такую рекурсию называют хвостовой.

Хвостовая рекурсия может показаться простой идеей, но у нее есть интересное свойство: ее можно оптимизировать с помощью компилятора или интерпретатора и заставить работать так же эффективно, как итеративное решение, например, на основе цикла. Подобная оптимизация возможна благодаря структурированной форме хвостовой рекурсии, которая следует шаблону, аналогичному шаблону циклов.

В результате эта оптимизация может привести к значительному повышению производительности рекурсивных алгоритмов, особенно при работе с большими входными данными или сложными вычислениями.

Вернемся к предыдущей функции вычисления факториала, но теперь перепишем ее в стиле хвостовой рекурсии:

```
def factorial(n, accumulator=1):
    if n == 0:
        return accumulator
    else:
        return factorial(n-1, n * accumulator)
```

В этой версии функции `accumulator` используется для хранения результата вычисления факториала на каждом шаге, а рекурсивный вызов `factorial(n-1, n * accumulator)` является последней операцией, выполняемой в функции. Это пример хвостовой рекурсии.

Не все языки программирования или среды могут оптимизировать хвостовую рекурсию. В тех случаях, когда это возможно, например в некоторых реализациях Scheme и других функциональных языках, функции хвостовой рекурсии могут работать эффективнее их аналогов, не имеющих хвостовой рекурсии. В других языках, таких как Python или Java, они не дают преимущества в производительности, и программист должен помнить о потенциальных ошибках переполнения стека.

Понятие хвостовой рекурсии помогает представить более широкую тему оптимизации рекурсивных алгоритмов — сложную и увлекательную область информатики.

4.5. Практические задачи

Представленные ниже задачи помогут вам получить практическое представление о различных типах алгоритмов, особенностях их работы и о том, где их можно применять. Попробуйте решить все задачи и сравните свои решения с нашими, чтобы увидеть разницу между этими типами алгоритмов. Удачи!

Задача 1. Двоичный поиск (разделяй и властвуй)

Даны отсортированный список из n целых чисел и искомое число x. Напишите функцию, которая находит местоположение x в списке с помощью алгоритма двоичного поиска. Помните, что двоичный поиск эффективно делит задачу пополам на каждом шаге.

Решение

```
def binary_search(list, x):
    low = 0
    high = len(list) - 1

    while low <= high:
        mid = (high + low) // 2
```

```
        if list[mid] < x:
            low = mid + 1
        elif list[mid] > x:
            gigh = mid - 1
        else:
            return mid
    return -1
```

Задача 2. Размен монет (жадный алгоритм)

Даны список номиналов монет и значение *n*. Напишите функцию, вычисляющую минимальное количество монет, необходимое для сдачи *n*. Используйте жадный алгоритм, который всегда выбирает монету с наибольшим номиналом, меньшим или равным оставшейся сумме сдачи. Проверьте свою реализацию на различных наборах номиналов монет, например номиналах, являющихся стандартными для США или Европы.

Решение

```
def coin_change(coins, n):
    coins.sort(reverse=True)
    count = 0
    for coin in coins:
        while n >= coin:
            n -= coin
            count += 1
        if n == 0:
            return count
    return count
```

Задача 3. Числа Фибоначчи (динамическое программирование)

Напишите функцию для вычисления n-го числа Фибоначчи. Ряд Фибоначчи определяется как $F(0) = 0$, $F(1) = 1$ и $F(n) = F(n - 1) + F(n - 2)$ для $n > 1$. Сначала напишите простое рекурсивное решение, а затем оптимизируйте его, применив метод динамического программирования, в котором вычисленные значения сохраняются и используются повторно.

Решение

```
def fibonacci(n, computed={0: 0, 1: 1}):
    if n not in computed:
        computed[n] = fibonacci(n-1, computed) + fibonacci(n-2, computed)
    return computed[n]
```

Задача 4. Сумма натуральных чисел (рекурсивный алгоритм)

Напишите функцию, которая вычисляет сумму всех натуральных чисел от 1 до *n* с помощью рекурсивного алгоритма. Сравните рекурсивное решение с итеративным и обсудите различия.

Решение

```
def recursive_sum(n):
    if n <= 1:
        return n
    else:
        return n + recursive_sum(n-1)

def iterative_sum(n):
    return n * (n+1) // 2
```

Задача 5. Быстрая сортировка (разделяй и властвуй)

Реализуйте алгоритм QuickSort, который сортирует массив целых чисел. Это алгоритм из категории «разделяй и властвуй». Он выбирает опорный элемент и разбивает вокруг него массив.

Решение

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

Задача 6. Реализация стека с помощью рекурсии (рекурсивный алгоритм)

Реализуйте структуру данных стека, используя рекурсию. Ваш стек должен иметь функции `push`, `pop` и `peek`.

Решение

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if len(self.stack) < 1:
            return None
        return self.stack.pop()

    def peek(self):
        if self.stack:
            return self.stack[-1]
        return None
```

В следующей главе мы рассмотрим более сложные типы алгоритмов и их применение при решении реальных задач.

Резюме

В этой главе мы познакомились с крайне важной темой — основными типами алгоритмов. Их четыре: «разделяй и властвуй», жадные алгоритмы, алгоритмы динамического программирования и рекурсивные алгоритмы.

Сначала мы исследовали алгоритмы «разделяй и властвуй», стратегия которых выглядит так: сложная задача разбивается на более мелкие и управляемые подзадачи, они решаются по отдельности, а затем эти решения объединяют, чтобы получить решение исходной задачи. Вы узнали, что классическим примером такого подхода является алгоритм двоичного поиска, который многократно делит отсортированный список пополам, чтобы найти определенное значение. Благодаря своей эффективности алгоритмы этого типа часто используются в операциях сортировки и поиска, при обработке баз данных и во многих других приложениях.

Затем мы перешли к жадным алгоритмам. Согласно их стратегии, на каждом шаге всегда делается локально оптимальный выбор из расчета на то, что этот выбор приведет к глобальному оптимуму. Мы рассмотрели задачу размена монет как практическое применение жадного алгоритма, обсудив

его сильные стороны и потенциальные недостатки, а также выяснили, что он не всегда гарантирует оптимальное решение.

Далее мы исследовали алгоритмы динамического программирования. Этот подход применяется, когда задача состоит из множества перекрывающихся подзадач. Используя найденные ранее решения подзадач, алгоритм ДП позволяет избежать лишней работы и добиться значительной эффективности. Практическим примером, иллюстрирующим способность такого алгоритма преобразовать экспоненциально сложную задачу в линейную, послужил ряд Фибоначчи.

Под конец главы мы углубились в рекурсивные алгоритмы — метод, в котором решение задачи зависит от решений более мелких экземпляров одной и той же задачи. Мы обсудили важность определения базового случая в рекурсивных решениях и в качестве примера рассмотрели задачу вычисления суммы натуральных чисел. Мы также затронули понятие хвостовой рекурсии и обсудили ее возможности по оптимизации в некоторых языках программирования.

Чтобы вы могли закрепить новые знания, мы предложили вам решить несколько задач. Целью их было помочь вам получить практический опыт реализации и понять основные типы алгоритмов, акцентировать ваше внимание на их реальном применении и показать, насколько важно выбирать правильный алгоритм для конкретной задачи.

Итак, вы познакомились с базовыми типами алгоритмов. Это знание послужит вам основой, благодаря которой вы сможете решать более сложные алгоритмические задачи, представленные в следующих главах.

Глава 5
АЛГОРИТМЫ ПОИСКА

Алгоритмы поиска — неотъемлемая часть многих операций в информатике и повседневной жизни. Они позволяют быстро и эффективно находить информацию, будь то поиск ключевого слова в документе или контакта в телефоне либо получение веб-страниц из поисковых систем на основе введенного нами критерия.

Алгоритмы поиска — обширная область программирования, в которой используется множество методов. Каждый из них имеет сильные и слабые стороны и конкретные варианты применения. Глубина понимания данной темы, умение анализировать алгоритмы поиска и выбирать правильный могут существенно повлиять на эффективность программ, особенно работающих с большими наборами данных.

В этой главе мы исследуем различные алгоритмы поиска, начиная с одного из самых простых, но действенных методов — линейного поиска. Этот алгоритм прост в реализации и может быть полезен во многих ситуациях. Однако у него есть ограничения, и мы обсудим его альтернативные варианты, более пригодные для конкретных ситуаций. К концу главы вы получите четкое представление о различных алгоритмах поиска, особенностях их работы и случаях, в которых лучше всего их использовать.

5.1. Линейный поиск

Линейный поиск, также известный как *последовательный поиск*, — это простой, но эффективный метод поиска определенного значения в списке. Он последовательно проверяет каждый элемент списка, пока не найдет совпадение или не достигнет конца списка. Этот метод особенно полезен, когда количество элементов в списке не очень велико.

Кроме того, поскольку линейный поиск не требует сортировки списка, его можно использовать вместе с неотсортированными данными. Одно из преимуществ этого алгоритма — его легко реализовать с помощью цикла,

поэтому его часто используют люди, которые только начинают изучать программирование. Более того, простота его реализации облегчает отладку, что может быть полезно при работе над крупными и сложными программами.

Еще одно преимущество линейного поиска — его легко адаптировать под конкретные потребности. Например, его можно использовать для поиска первого появления значения в списке или всех вхождений искомого значения. Такая гибкость делает его универсальным инструментом, который можно применять для решения широкого спектра задач, как простых, так и сложных.

Проиллюстрируем вышесказанное примером кода на Python:

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i # вернуть индекс найденного элемента
    return -1 # вернуть -1, если элемент не найден

# Тестирование функции
arr = [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]
target = 110

result = linear_search(arr, target)
if result != -1:
    print("Элемент находится по индексу", str(result))
else:
    print("Элемента нет в массиве")
```

В этом примере определяется функция с именем `linear_search`, которая принимает массив и целевое значение. Она проверяет каждый элемент массива, пока не найдет элемент, соответствующий искомому. Найдя целевое значение, она возвращает его индекс в массиве; в противном случае возвращается `-1`, указывающее, что целевое значение отсутствует в массиве.

Важно отметить, что линейный поиск — не самый эффективный алгоритм, когда речь заходит о больших наборах данных. Фактически в худшем случае (когда искомое значение находится в конце списка или вообще отсутствует) ему придется проверить все элементы в списке, что может занять много времени и привести к временной сложности $O(n)$, где n — размер списка.

Несмотря на это, линейный поиск остается ценным алгоритмом, который важно знать и понимать, поскольку он служит основой для более сложных алгоритмов поиска, таких как двоичный и интерполяционный поиск.

Стоит упомянуть конкретные сценарии, в которых линейный поиск может быть особенно полезен.

- **Небольшие списки.** Линейный поиск обычно считается менее эффективным, чем более сложные алгоритмы, такие как двоичный поиск или поиск в хеш-таблицах, но короткие списки может обрабатывать лучше всего.

  Это связано с тем, что накладные расходы на сортировку списка или построение хеш-таблицы, совершенно необходимые для двоичного поиска или поиска в хеш-таблице соответственно, могут перевесить преимущества сложных алгоритмов, когда списки невелики. Фактически линейный поиск по-прежнему широко используется в ряде приложений, где небольшие списки являются нормой, например во встраиваемых системах или некоторых инструментах обработки данных.

  Линейный поиск проще в реализации и более понятен для тех, кто плохо знаком с программированием или не имеет опыта работы в области информатики. Поэтому для больших списков линейный поиск не самый эффективный вариант, а вот с маленькими списками он справляется отлично.

- **Неупорядоченные данные.** Как упоминалось выше, линейный поиск не требует сортировки входных данных. Когда они изначально не упорядочены и их сортировка нецелесообразна (например, из-за ограничений памяти или динамического характера данных), вполне можно использовать линейный поиск.

  Это связано с тем, что данный алгоритм последовательно просматривает входные данные, пока не найдет целевой элемент. Это свойство делает его особенно полезным, когда входные данные не организованы в определенном порядке, поскольку алгоритм все равно найдет нужный элемент, ничего дополнительно не обрабатывая или не сортируя.

  Кроме того, линейный поиск в небольших наборах данных часто выполняется быстрее, чем другие алгоритмы поиска. Это связано с отсутствием накладных расходов на сортировку. Однако важно отметить, что большие наборы данных линейный поиск может обрабатывать неэффективно из-за его временной сложности, $O(n)$. В таких случаях стоит использовать другие алгоритмы поиска, такие как двоичный поиск или поиск в хеш-таблице.

- **Последовательный доступ к памяти.** Современные процессоры имеют сложную систему кэширования, и иногда последовательный доступ к памяти (как при линейном поиске) происходит быстрее, чем перемещения

в разные концы списка (как при двоичном поиске). Однако это во многом зависит от конкретной архитектуры системы, а также характера и размера данных.

Более того, эффективность последовательного доступа к памяти может меняться в зависимости от приложения и типа данных, к которым осуществляется доступ. Например, доступ к последовательным данным может быть более эффективным при работе с большими смежными блоками памяти, как при чтении или записи файлов. В то же время произвольный доступ может оказаться более эффективным при работе с меньшими объемами данных или при поиске определенных фрагментов информации в большем наборе данных.

Важно отметить еще и то, что последовательный доступ к памяти подойдет не везде. В некоторых случаях затраты на его поддержание могут перевесить преимущества, особенно в системах, использующих сложные алгоритмы кэширования. Более того, на эффективность последовательного доступа к памяти может влиять и конкретная реализация алгоритма. Поэтому, выбирая стратегию доступа, важно учесть конкретные требования приложения.

- **Потоковая передача данных, или передача данных в реальном времени.** Линейный поиск можно использовать, когда данные передаются в режиме реального времени или полный набор данных недоступен на момент поиска, поскольку он не требует наличия всего набора данных, в отличие от алгоритмов двоичного поиска или поиска в хеш-таблицах.

  Линейный поиск — это алгоритм поиска определенного значения в списке, последовательности или массиве путем последовательной проверки каждого элемента, пока не будет найдено совпадение или достигнут конец списка. Поэтому линейный поиск особенно полезен в случаях, когда данные не отсортированы или ожидается, что пространство поиска будет ограничено несколькими значениями.

  Кроме того, линейный поиск легко распараллелить — его можно разделить на более мелкие задачи и выполнять их одновременно на нескольких процессорах, что ускорит процесс поиска. Однако стоит отметить, что на больших или отсортированных наборах данных линейный поиск может работать медленнее других алгоритмов, таких как двоичный поиск.

Очень важно помнить, что выбор алгоритма для конкретной задачи всегда зависит от конкретных требований и ограничений. В определенных обстоятельствах линейный поиск может оказаться отличным вариантом. Однако

при работе с большими наборами данных предпочтительнее использовать более эффективные алгоритмы поиска.

Понимая идею линейного поиска, вы сможете изучать более сложные поисковые алгоритмы. В последующих разделах мы рассмотрим те из них, которые могут более эффективно обрабатывать значительные наборы данных, а пока продолжим обсуждение линейного поиска.

5.1.1. Ограничения линейного поиска

Несмотря то что линейный поиск прост и в ряде случаев полезен, у него есть определенные ограничения.

Масштабируемость

Линейный поиск — широко используемый алгоритм, особенно когда дело касается обработки небольших наборов данных. Однако он не самый эффективный — по мере увеличения объема данных может замедляться. Это связано с тем, что алгоритм последовательно проверяет каждый элемент, затрачивая много времени и ресурсов.

Это может быть серьезной проблемой в приложениях, обрабатывающих большие объемы данных, таких как анализ больших данных и машинное обучение. Поэтому при обработке больших объемов важно подумать о возможности применения альтернативных алгоритмов. Например, двоичный поиск — более эффективный алгоритм, позволяющий значительно сократить время поиска в больших наборах данных.

Он работает путем разделения набора данных на более мелкие сегменты и поиска в одном из них целевого элемента, сокращая время поиска вдвое с каждой итерацией. Таким образом, алгоритм линейного поиска полезен при работе с небольшими наборами данных, но может не подойти для больших наборов, и в подобных случаях желательно рассмотреть альтернативные алгоритмы, которые могут улучшить масштабируемость и эффективность приложения.

Скорость

Временная сложность линейного поиска равна $O(n)$, то есть затрачиваемое им время растет линейно по мере увеличения объема входных данных. Это не идеальный вариант при работе с большими наборами данных, где

более эффективные алгоритмы могут справляться с той же задачей быстрее. Например, алгоритм двоичного поиска имеет временную сложность $O(\log n)$, что позволяет выполнять поиск быстрее.

Другие, более сложные алгоритмы, такие как поиск в хеш-таблицах, могут работать еще быстрее. По мере увеличения объема данных разница в скорости между этими алгоритмами становится все заметнее. Поэтому важно учитывать размер набора данных при выборе подходящего алгоритма поиска.

Отсутствие оптимизации

Линейный поиск — простой и довольно эффективный алгоритм поиска определенного элемента в списке. Однако у него есть ряд ограничений, которые могут помешать его работе в некоторых ситуациях. Одно из таких ограничений — отсутствие оптимизации. В отличие от более сложных алгоритмов поиска, линейный не использует никакую информацию об упорядоченности или структуре набора данных в целях оптимизации поиска.

По этой причине он вынужден последовательно проверять каждый элемент в списке, пока не найдет нужный, что может быть неэффективно при работе с большими наборами данных. Несмотря на это, линейный поиск остается ценным инструментом в арсенале программиста, особенно в ситуациях обработки маленьких наборов данных, поскольку простота реализации облегчает его использование.

В следующих разделах мы рассмотрим более эффективные алгоритмы поиска, решающие некоторые из описанных выше проблем. Однако помните: у каждого алгоритма есть свои компромиссы, и выбор во многом зависит от решаемой задачи. Важно понимать сильные и слабые стороны каждого алгоритма, чтобы принять обоснованное решение при выборе лучшего подхода в любой конкретной ситуации.

5.2. Двоичный поиск

Двоичный поиск — это алгоритм, более эффективный, чем линейный поиск, при соблюдении определенных условий. Он следует принципу «разделяй и властвуй», о котором мы подробно говорили в главе 4.

Чтобы лучше понять этот алгоритм, подробно рассмотрим, как он работает. Сначала двоичный поиск проверяет средний элемент отсортированного

списка. Если тот совпадает с искомым значением, это означает, что поиск увенчался успехом и процесс можно прекратить. Если целевое значение меньше среднего элемента, то можно предположить, что оно отсутствует в правой половине списка. В результате процесс поиска продолжится только в левой половине. В то же время если целевое значение больше среднего элемента, то можно с уверенностью предположить, что оно отсутствует в левой половине списка. Следовательно, процесс поиска продолжится только в правой половине.

Процесс сравнения и деления пространства поиска продолжается до тех пор, пока не будет найдено целевое значение или пространство поиска не станет пустым. С каждой итерацией пространство поиска уменьшается вдвое, что позволяет алгоритму двоичного поиска быстро сузить границы поиска и найти целевое значение намного быстрее, чем это сделал бы алгоритм линейного поиска.

Как и было обещано, теперь рассмотрим пример работы двоичного поиска и некий код, который поможет проиллюстрировать этот процесс.

Предположим, у нас есть отсортированный список чисел:

```
numbers = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
```

и нам нужно найти число 23.

1. Сначала проверим средний элемент списка, то есть 16. Поскольку 23 > 16, мы знаем, что число 23 должно находиться в правой половине списка.
2. Далее проверим середину правой половины — число 38. Поскольку 23 < 38, мы знаем, что число 23 должно находиться в левой половине оставшегося подсписка (`[23, 38]`).
3. Затем проверим середину списка `[23, 38]` — число 23. Поскольку 23 == 23, то мы считаем, что искомая цель найдена и поиск окончен.

Теперь посмотрим, как реализовать этот алгоритм на Python:

```
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid # Элемент найден, вернуть его индекс
```

```
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1 # Элемент не найден

# Тестирование функции
numbers = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
print(bunary_search(numbers, 23)) # Выведет 5
```

В этом фрагменте кода на Python мы определяем функцию `binary_search`, которая принимает отсортированный список `arr` и целевое значение `target`. Первым делом функция присваивает указателям `left` и `right` индексы первого и последнего элементов списка соответственно. Затем входит в цикл, который продолжается до тех пор, пока указатели `left` и `right` не встретятся.

В каждой итерации цикла вычисляется индекс среднего элемента `mid`. Если этот элемент равен целевому, то функция возвращает его индекс. Если искомое значение больше, то в указатель `left` записывается индекс `mid + 1`. Если меньше, то в указатель `right` записывается индекс `mid - 1`. Если цикл завершился, так и не найдя целевого значения, то функция возвращает `-1`, тем самым сообщая, что цель не найдена.

А теперь оценим производительность алгоритма двоичного поиска и сравним его с другими поисковыми алгоритмами.

Данный алгоритм демонстрирует значительное улучшение временной сложности по сравнению с линейным поиском. Каждое сравнение сокращает пространство поиска вдвое. Следовательно, чтобы найти искомое значение в списке из n элементов или убедиться в его отсутствии, в худшем случае потребуется проверить $\log_2(n)$ элементов.

Учитывая эту логарифмическую зависимость между размером списка и количеством шагов, мы говорим, что двоичный поиск имеет временную сложность $O(\log n)$. Помните, что здесь подразумевается логарифм по основанию 2, но в нотации «O большое» эта деталь обычно игнорируется, поскольку нас больше интересуют темпы роста, а не конкретные величины.

С точки зрения пространственной сложности алгоритм двоичного поиска тоже показывает себя с лучшей стороны: ему требуется постоянный объем дополнительного пространства для хранения переменных `left`, `right` и `mid`, независимо от размера списка. Соответственно, пространственная сложность равна $O(1)$.

Благодаря улучшенной временной сложности и постоянной пространственной сложности двоичный поиск больше подходит для поиска в отсортированных списках, особенно когда размер списка велик. Однако требование сортировки списка — это компромисс, который необходимо учитывать.

Имея представление о сложности такого алгоритма, как двоичный поиск, вы сможете сделать обоснованный выбор решения для конкретных задач, особенно когда имеют место ограничения по времени и потребляемой памяти.

5.3. Хеширование и хеш-таблицы

Хеширование — один из важнейших методов, обеспечивающих быстрый доступ к данным, хранящимся в памяти. Он основан на простой, но эффективной концепции: отображении фактических значений в определенные места в памяти, где эти значения можно быстро и эффективно найти. Основная идея хеширования заключается в использовании хеш-функции, которая преобразует входные данные, также называемые ключом, в уникальный индекс, соответствующий ячейке памяти, в которой хранятся данные. Это означает, что после того как данные будут хешированы и сохранены, их можно мгновенно получить, если использовать ту же хеш-функцию для вычисления индекса.

Хеш-функция — это математическая функция, которая принимает входное значение, обычно строку или число, и возвращает результат фиксированного размера — хеш-значение. Затем оно используется в качестве индекса для доступа к данным в памяти. Идеальная хеш-функция должна равномерно распределять данные по памяти, чтобы избежать коллизий, когда двум ключам назначается один и тот же индекс. Однако найти идеальную хеш-функцию — непростая задача, и для обработки коллизий используются различные методы, такие как объединение в цепочку или открытая адресация.

Объединение в цепочку — это метод сохранения нескольких значений с одним и тем же индексом в форме связанного списка. При возникновении коллизии новое значение просто добавляется в конец связанного списка. Открытая адресация, в свою очередь, предполагает поиск следующего доступного индекса при возникновении коллизии. Это можно сделать с помощью различных алгоритмов, таких как линейное или квадратичное зондирование. Более подробно об этих методах мы поговорим чуть позже.

Таким образом, хеширование — метод, позволяющий эффективно извлекать данные в процессе вычислений. Он основан на идее использования хеш-функции для отображения данных в определенную ячейку памяти, а для обработки коллизий предназначены другие методы. Понимая суть хеширования и имеющиеся требования, программисты могут создавать быстрое и эффективное программное обеспечение, способное обрабатывать большие объемы данных в режиме реального времени.

Хеш-таблица — фундаментальное понятие в информатике и структура данных, широко используемая для хранения и извлечения данных. Это инструмент, благодаря которому можно получить быстрый и эффективный доступ к данным с помощью процесса хеширования. Хеширование преобразует ключ в индекс или адрес массива сегментов или слотов, хранящего значение. Это означает, что к данным можно быстро получить доступ и при этом не придется выполнять поиск по всему набору данных.

Одно из преимуществ использования хеш-таблиц — возможность хранить пары «ключ — значение», что может пригодиться во многих приложениях. Например, хеш-таблицу можно использовать для хранения сведений о человеке, указывая его имя в качестве ключа, а информацию о нем, такую как его адрес, номер телефона и адрес электронной почты, — в качестве значения. Это позволит быстро получить данные о человеке, просто выполнив поиск по его имени.

Еще одно преимущество хеш-таблиц — способность обрабатывать большие объемы данных. Хеш-таблицы используют хеш-функцию для вычисления индекса в массиве сегментов или слотов, поэтому даже большие наборы данных можно эффективно хранить и выполнять в них поиск. Кроме того, размер хеш-таблиц можно изменять динамически, а это означает, что они могут увеличиваться или уменьшаться по мере необходимости в соответствии с объемом хранимых данных.

В целом хеш-таблица — важный инструмент информатики, который используется в самых разных приложениях. Независимо от объема данных, хеш-таблицы позволят вам быстро и эффективно сохранять и извлекать данные.

Вот пример реализации простой хеш-функции и простой хеш-таблицы на Python:

```
# Определить простую хеш-функцию
def simple_hash(key):
    return key % 10

# Инициализировать хеш-таблицу как список с десятью элементами
hash_table = [None] * 10
```

```
# Добавить некоторые данные
key = 35
value = "Apple"

# Вычислить индекс
index = simple_hash(key)

# Сохранить значение value в хеш-таблицу
hash_table[index] = value

print(hash_table)
```

Этот код выведет:

```
[None, None, None, None, None, 'Apple', None, None, None, None]
```

В этом примере, чтобы определить, где сохранить значение `"Apple"`, мы использовали простую хеш-функцию `key % 10`. Ключ `key` равен `35`, а 35 % от 10 равно 5, поэтому значение `"Apple"` сохраняется в элементе списка с индексом 5.

Но имейте в виду, что это очень простой пример, созданный в иллюстративных целях. На практике хеш-функции могут быть гораздо более сложными, а хеш-таблицы обязательно должны содержать методы обработки коллизий, а также методы добавления, удаления и извлечения данных.

Помните: эффективность хеш-таблицы сильно зависит от хеш-функции и коэффициента перегрузки (отношения количества элементов к количеству слотов). При правильной реализации хеш-таблицы операции поиска, вставки и удаления могут показывать временную сложность O(1).

Хеш-таблицы используются во множестве приложений, таких как индексирование баз данных, кэширование, хранение паролей и многое другое. Возможность быстрого доступа к данным по ключу делает эти таблицы невероятно полезными в ситуациях, когда быстрый доступ имеет решающее значение.

5.3.1. Коллизии

Коллизии — обычное явление в хеш-функциях, возникающее, когда два разных входных значения отображаются в одно и то же выходное. Теоретически хеш-функции должны быть детерминированными и для разных входных значений возвращать разные результаты, но на практике возможны коллизии, которые могут вызвать проблемы.

Разрешить эти коллизии помогают разные методы, такие как объединение в цепочку, открытая адресация и двойное хеширование. Объединение в цепочку предполагает сохранение значений, получивших один и тот же индекс, в виде цепочки, а открытая адресация заключается в поиске следующего доступного индекса.

Двойное хеширование — более сложный метод, в котором для разрешения коллизий используются две хеш-функции. Имея представление о разных методах разрешения коллизий, можно создавать весьма эффективные хеш-функции для широкого спектра приложений.

Рассмотрим некоторые стратегии разрешения коллизий более подробно.

Объединение в цепочку

Объединение в цепочку — это метод разрешения коллизий в хеш-таблицах. При его использовании каждый индекс в таблице фактически представляет связанный список, содержащий все ключи, хеш-значения которых совпадают с этим индексом. Когда возникает коллизия, пара «ключ — значение» просто добавляется в конец списка, соответствующего индексу.

Этот метод позволяет обрабатывать коллизии более эффективно, при этом постоянное время поиска в хеш-таблицах будет сохраняться. Найти значение можно так: сначала нужно хешировать ключ, чтобы найти индекс, а затем просмотреть элементы связанного списка, соответствующего этому индексу, чтобы найти целевое значение. Преимущество этого подхода заключается в простоте реализации и предсказуемости производительности в худшем случае, вследствие чего он часто используется для реализации хеш-таблиц.

Вот пример:

```
# Пример реализации хеш-таблицы на Python
# с использованием метода объединения в цепочку
hash_table = [[] for _ in range(10)]

def insert(hash_table, key, value):
    hash_key = hash(key) % len(hash_table)
    key_exists = False
    bucket = hash_table[hash_key]
    for i, kv in enumerate(bucket):
        k, v = kv
        if key == k:
            key_exists = True
            break
```

```
    if key_exists:
        bucket[i] = ((key, value))
    else:
        bucket.append((key, value))

# Вставить несколько значений
insert(hash_table, 10, 'Apple')
insert(hash_table, 25, 'Banana')
insert(hash_table, 20, 'Cherry')
```

В этом случае оба ключа, `10` и `20`, будут хешированы в один и тот же индекс (`0`), поэтому для обработки коллизии новая пара «ключ — значение» будет добавлена в конец списка, соответствующего индексу.

Открытая адресация

Открытая адресация — один из методов разрешения коллизий в хеш-таблицах. В соответствии с ним все пары «ключ — значение» хранятся в самой хеш-таблице, а при возникновении коллизии хеш-функция ищет следующий доступный слот в таблице. Существуют разные способы поиска следующего пустого слота, называемые последовательностями зондирования.

Один из таких способов — линейное зондирование, когда хеш-функция последовательно проверяет каждый слот в массиве, пока не встретит первый незанятый. Другой способ — это квадратичное зондирование, когда хеш-функция проверяет слоты, совершая переходы на всё бóльшие расстояния. Наконец, еще одна последовательность зондирования — двойное хеширование, когда одна хеш-функция использует вторую для определения последовательности проверок.

Открытая адресация может потребовать больше времени, чем объединение в цепочку, но при определенных условиях способна обеспечить более высокую производительность.

Рассмотрим пример:

```
# Пример реализации хеш-таблицы на Python, использующей
# линейное зондирование для разрешения коллизий
hash_table = [None] * 10

def insert(hash_table, key, value):
    hash_key = hash(key) % len(hash_table)
    while hash_table[hash_key] is not None:
```

```
        hash_key = (hash_key + 1) % len(hash_table)
    hash_table[hash_key] = value

# Вставить несколько значений
insert(hash_table, 10, 'Apple')
insert(hash_table, 25, 'Banana')
insert(hash_table, 20, 'Cherry')
```

В этом случае если два ключа отображаются в один и тот же индекс, то второй ключ помещается в следующий доступный слот.

На первый взгляд, хеширование и хеш-таблицы могут показаться простыми, но в действительности они скрывают под собой массу сложностей. Однако понимать особенности этих структур важно любому программисту, поскольку они представляют собой эффективный способ обработки данных.

Один из наиболее важных аспектов реализации хеш-таблиц — выбор подходящей хеш-функции. Чтобы получить максимально эффективную хеш-таблицу, следует очень тщательно выбирать хеш-функцию, чтобы генерируемые ею ключи равномерно распределялись по массиву.

Она должна стремиться свести к минимуму коллизии, ухудшающие производительность хеш-таблицы. Более того, выбор хеш-функции зависит от типа данных, хранящихся в хеш-таблице.

Например, если в хеш-таблице хранятся данные строго определенного типа, то оптимизировать ее производительность можно с помощью определенной хеш-функции. Поэтому выбор подходящей хеш-функции — важный шаг в реализации хеш-таблицы.

Хорошая хеш-функция должна обладать следующими свойствами.

1. **Детерминированность.** Для одних и тех же входных данных всегда должен возвращаться один и тот же результат. Таким образом можно обеспечить согласованность и предсказуемость.
2. **Быстрое вычисление хеш-значения.** Хеш-функция должна максимально быстро вычислять хеш для любого входного значения, чтобы обеспечить высокую общую производительность.
3. **Равномерное распределение.** Хеш-функция должна равномерно распределять ключи по массиву, то есть каждый индекс в массиве должен быть равновероятным. Это свойство поможет предотвратить кластеризацию

значений в определенной области и избежать коллизий, снижающих общую эффективность хеш-таблицы и замедляющих поиск.

4. **Низкая вероятность возникновения коллизий.** Коллизии неизбежны, однако хорошая хеш-функция должна стремиться минимизировать их. Низкая частота коллизий способствует общей высокой эффективности хеш-таблицы и предотвращает снижение производительности, вызванное чрезмерно большим количеством коллизий.
5. **Надежность.** Хеш-функция должна обрабатывать широкий диапазон входных данных и для каждого входного значения создавать уникальный хеш. Это необходимо для того, чтобы хеш-таблица могла обрабатывать разные типы данных, не нанося ущерб производительности и эффективности.
6. **Безопасность.** В некоторых случаях важно, чтобы хеш-функция была безопасной и устойчивой к атакам. Например, в криптографии хеш-функции используются для проверки целостности данных и предотвращения их подделки. Безопасная хеш-функция должна предусматривать возможность противостояния таким атакам, как коллизии, атаки методом поиска прообразов и атаки на основе парадокса «день рождения» (birthday attacks).

Вот пример простой реализации хеш-функции:

```
def hash_function(key):
    return key % 10
```

Эта хеш-функция просто возвращает остаток от деления ключа `key` на размер массива (в данном случае 10). Вычисления выполняются очень быстро, но ключи могут распределяться неравномерно, особенно если в них наблюдаются некоторые закономерности.

Обратите внимание: описанные выше принципы составляют лишь самые основы. Хеширование — обширная область информатики, в которой продолжаются активные исследования и создаются все более сложные хеш-функции, стратегии разрешения коллизий и приемы их применения.

Прелесть хеш-таблиц в том, что они поддерживают связь между ключами и значениями, подобно словарям, и позволяют очень быстро (в идеале за постоянное время) выполнять операции поиска, добавления и удаления записей.

5.4. Практические задачи

Предложенные ниже задачи позволят вам применить на практике ваши новые знания об алгоритмах поиска. Решения приводятся тут же, но я бы посоветовал сначала попробовать найти их самостоятельно и только потом заглядывать в предложенные нами. Желаю получить максимум удовольствия от программирования!

Задача 1. Линейный поиск

Напишите на Python функцию, реализующую алгоритм линейного поиска. Функция должна принимать список и искомое значение и возвращать индекс искомого значения в списке, если оно найдено, или `-1` в противном случае.

Решение

```
def linear_search(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
    return -1
```

Задача 2. Двоичный поиск

Напишите на Python функцию, реализующую алгоритм двоичного поиска. Функция должна принимать отсортированный список и искомое значение и возвращать индекс искомого значения в списке, если оно найдено, или `-1` в противном случае.

Решение

```
def binary_search(lst, target):
    left, right = 0, len(lst) - 1
    while left <= right:
        mid = (left + right) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            left = mid +1
        else:
            right = mid - 1
    return -1
```

Задача 3. Хеширование

Предположим, вы реализовали хеш-таблицу, обрабатывающую коллизии путем объединения конфликтующих ключей в цепочку. Теперь вам даны ключи: 10, 22, 31, 4, 15, 28, 17, 88, 59. Напишите на Python функцию построения хеш-таблицы, которая использует хеш-функцию `key mod 10`.

Решение

```
def build_hash_table(keys):
    hash_table = [[] for _ in range(10)] # Инициализировать таблицу
                                         # как список пустых списков
    for key in keys:
        hash_key = key % 10 # Вычислить хеш ключа
        hash_table[hash_key].append(key) # Вставить ключ
                                         # в соответствующий список
    return hash_table

keys = [10, 22, 31, 4, 15, 28, 17, 88, 59]
print(build_hash_table(keys))
```

Задача 4. Сравнение эффективности двоичного и линейного поиска

Дан список из 1000 элементов. В какой момент (с какого количества элементов) двоичный поиск начинает выполняться быстрее, чем линейный? Объясните свои рассуждения.

В следующей главе мы углубимся в алгоритмы сортировки — еще один фундаментальный аспект алгоритмического мышления. Надеюсь, вы с нетерпением ждете продолжения нашего путешествия!

Резюме

В данной главе мы подробно рассмотрели алгоритмы поиска — важнейший раздел информатики и программирования. Мы обсуждали теоретические основы различных алгоритмов поиска, а также предоставили вам возможность получить опыт решения практических задач.

Мы начали с самого простого из алгоритмов поиска — линейного поиска. Этот простой, но важный алгоритм составляет основу многих более сложных

алгоритмов и является идеальной отправной точкой для данной главы. Линейный поиск последовательно проверяет каждый элемент в списке, пока не найдет совпадение с искомым значением или не достигнет конца списка. Мы отметили его простоту, а также недостаточную эффективность (обусловленную его временной сложностью $O(n)$), особенно заметную при работе с большими списками.

Затем мы перешли к двоичному поиску — гораздо более эффективному алгоритму, работающему с отсортированными списками. Постоянно деля список пополам и определяя, в какой половине продолжить поиск, двоичный поиск снижает временную сложность до $O(\log n)$. Однако такая экономия времени достигается за счет требования к упорядоченности списка, и данный аспект необходимо учитывать при выборе алгоритма.

Далее мы изучили хеширование и хеш-таблицы, позволяющие эффективно хранить и извлекать данные. Этот метод основан на применении хеш-функции, преобразующей искомое значение (ключ) в хеш-код, определяющий индекс элемента списка, в котором хранится соответствующая запись. Этот механизм позволяет реализовать операции поиска, добавления и удаления с временной сложностью $O(1)$.

Наконец, закрепить знания об алгоритмах поиска нам помогли несколько задач на тему реализации алгоритмов на Python и сравнения их эффективности. Эти задачи позволили вам получить практический опыт, не имея которого невозможно понять особенности работы этих алгоритмов.

В заключение отметим, что алгоритмы поиска являются важной частью инструментария любого программиста. Зная о достоинствах и недостатках этих алгоритмов, вы сможете выбрать тот, который решит ту или иную задачу наилучшим образом, вследствие чего значительно повысятся эффективность и производительность ваших программ. Далее в книге мы продолжим знакомиться с другими типами алгоритмов, расширяя ваши знания и навыки. В следующей главе мы сосредоточимся на алгоритмах сортировки — еще одной группе важнейших алгоритмов информатики. Так что продолжайте тренироваться и не отставайте от нас!

Глава 6
АЛГОРИТМЫ СОРТИРОВКИ

Сортировка является одной из важнейших задач в информатике и играет решающую роль во многих приложениях. Без нее поиск в больших объемах данных становится сложной и трудоемкой задачей. Представьте, что вы пытаетесь найти конкретную книгу в библиотеке, где тома на полках расставлены как попало, без всякой организации. Вам придется просмотреть все книги, пока вы не найдете то, что ищете. Однако если бы книги были отсортированы, например, по алфавиту, то вы смогли бы найти нужную книгу гораздо быстрее. Вот чем хороша сортировка!

В этой главе мы рассмотрим различные алгоритмы сортировки, начав с простых и постепенно переходя к более сложным. Мы обсудим идеи, лежащие в основе каждого алгоритма, их достоинства и недостатки, а также временную и пространственную сложности. Мы также рассмотрим примеры воплощения алгоритмов в программный код, которые помогут вам понять, как реализуется каждый алгоритм. К концу этой главы вы получите полное представление об этих алгоритмах и их применении, что позволит вам выбрать наиболее подходящий под ваши конкретные требования.

6.1. Пузырьковая сортировка

Начнем с алгоритма пузырьковой сортировки. Это один из простейших алгоритмов, понятный даже новичкам. С него хорошо начинать изучение логики сортировки, поскольку он служит основой для исследования более сложных алгоритмов.

Алгоритм пузырьковой сортировки многократно обходит сортируемый список, сравнивает каждую пару соседних элементов и меняет их местами, если они стоят в неправильном порядке. Алгоритм сортирует список, перемещая большие или меньшие элементы к концу или началу списка соответственно. Этот процесс повторяется до тех пор, пока список не будет отсортирован по возрастанию или убыванию.

Пузырьковая сортировка проста, но имеет некоторые ограничения. С точки зрения временной сложности пузырьковая сортировка — не самый эффективный алгоритм. Его средняя и худшая сложности равны O(n^2), где n — количество элементов в сортируемом списке. Это означает, что по мере увеличения длины сортируемого списка время выполнения алгоритма растет в квадратичной прогрессии, что делает его непригодным для сортировки больших наборов данных. Однако небольшие списки или списки, которые почти отсортированы, пузырьковая сортировка может обработать вполне эффективно.

Рассмотрим пример реализации пузырьковой сортировки на Python:

```
def bubble_sort(list):
    n = len(list)

    for i in range(n):
        # Определить флаг, который позволит функции закончить
        # сортировку раньше, если это возможно
        already_sorted = True

        for j in range(n - i - 1):
            if list[j] > list[j + 1]:
                # Поменять значения местами
                list[j], list[j + 1] = list[j + 1], list[j]
                # Присвоить флагу значение False, чтобы повторить цикл
                already_sorted = False

        # Если в последней итерации не встретилось ни одной пары,
        # элементы которой пришлось поменять местами,
        # значит, сортировка списка завершена
        if already_sorted:
            break

    return list
```

Эта реализация сортирует входной список в порядке возрастания. Внешний цикл выполняет обход всех элементов, а внутренний — сравнивает каждый элемент с соседним и меняет их местами, если они стоят не по порядку. Флаг `already_sorted` оптимизирует алгоритм, помогая функции завершиться раньше, если обнаружится, что список уже отсортирован.

Чтобы получить более глубокое понимание пузырьковой сортировки, желательно поэкспериментировать с различными входными данными и понаблюдать, как функция работает в разных условиях. Найдите время, чтобы изучить особенности поведения алгоритма и посмотреть, как различные наборы данных влияют на его производительность.

Помните: вы в самом начале вашего пути в мир алгоритмов сортировки. Есть много других интересных методов сортировки, каждый из которых имеет свои сильные и слабые стороны. Итак, продолжайте исследования и расширяйте свои познания об увлекательном мире информатики!

6.1.1. Использование пузырьковой сортировки

Алгоритм пузырьковой сортировки может быть эффективным в следующих случаях.

- **Небольшие наборы данных.** Как уже говорилось выше, пузырьковая сортировка имеет временную сложность O(n^2). Соответственно, обработка больших наборов данных с помощью этого алгоритма может оказаться не самой эффективной. Однако для небольших наборов данных пузырьковая сортировка вполне подойдет, так как проста и легка в реализации.

  Кроме того, стоит отметить, что пузырьковая сортировка может пригодиться в ситуациях, когда набор данных частично или почти полностью отсортирован, поскольку в этих случаях временная сложность может быть ниже. Таким образом, даже притом что этот алгоритм может не подойти для работы с большими наборами данных, всегда полезно помнить, что в некоторых ситуациях он все еще может быть полезен и заслуживает рассмотрения.

- **Почти отсортированные данные.** Пузырьковая сортировка — довольно эффективный алгоритм при работе с почти отсортированными данными. Если исходные данные в какой-то степени уже отсортированы, то алгоритм заметит это и прекратит выполнение раньше. Это связано с тем, что он предусматривает преждевременное завершение работы, если обнаружится, что в какой-то итерации ему не придется менять местами какие-либо элементы.

  Благодаря этому пузырьковая сортировка экономит время и вычислительные ресурсы, которые в противном случае были бы потрачены на ненужные итерации. Таким образом, если ваши исходные данные почти отсортированы и имеют лишь несколько элементов, стоящих не по порядку, то использование пузырьковой сортировки может стать быстрым и эффективным решением.

- **Изучение основных идей алгоритмов сортировки.** Алгоритм пузырьковой сортировки отлично подходит для учебных целей. Он прост и понятен и позволяет разобраться в том, как работают алгоритмы данного типа.

Один из подходов к обучению — подробные объяснения того, как работает пузырьковая сортировка. Пузырьковая сортировка — простой алгоритм, многократно меняющий местами соседние элементы, если они расположены в неправильном порядке. Этот процесс повторяется несколько раз, пока список не будет отсортирован. Объяснение особенностей работы этого алгоритма поможет учащимся лучше понять, как он работает и почему это эффективный способ сортировки данных.

Еще один подход к обучению — обсуждение ограничений пузырьковой сортировки. Этот алгоритм отлично подходит для учебных целей, но неэффективен при работе с большими наборами данных. Фактически его временная сложность равна $O(n\hat{}2)$, что делает его малопригодным для сортировки больших объемов данных. Помимо пузырьковой сортировки, существуют другие алгоритмы, такие как быстрая сортировка или сортировка слиянием, которые эффективно справляются с большими наборами данных и обычно используются в реальных приложениях.

В целом пузырьковая сортировка — отличный алгоритм для обучения, но важно понимать его ограничения, а также изучать другие алгоритмы сортировки. Это позволит учащимся более глубоко понять основные идеи алгоритмов сортировки и разобраться в особенностях их применения в различных контекстах.

Ниже перечислены ситуации, когда пузырьковую сортировку лучше не использовать.

- **Обработка больших наборов данных.** Пузырьковая сортировка — алгоритм сортировки, эффективно обрабатывающий маленькие наборы данных, но крайне неэффективный при работе с большими наборами. Это связано с тем, что пузырьковая сортировка имеет временную сложность $O(n\hat{}2)$ и обработка очень больших наборов данных требует много времени. Однако существуют и другие алгоритмы сортировки, такие как быстрая сортировка, сортировка слиянием или пирамидальная сортировка.

  Эти алгоритмы имеют временную сложность $O(n \log n)$ и могут действовать намного эффективнее и быстрее. Поэтому при работе с большими наборами данных важно рассмотреть возможность использования этих алгоритмов сортировки, чтобы гарантировать выполнение операций за разумное время.

- **Работа с приложениями, требующими высокой производительности.** В этом случае лучше выбрать алгоритм более эффективный, чем

пузырьковая сортировка. Например, вы можете использовать быструю сортировку, сортировку слиянием или пирамидальную сортировку.

Эти алгоритмы имеют разные характеристики, делающие их более подходящими для конкретных случаев использования. С одной стороны, быстрая сортировка весьма эффективна на практике и имеет меньшую пространственную сложность, чем сортировка слиянием, но может медленно обрабатывать уже отсортированные массивы. С другой стороны, сортировка слиянием стабильна и хорошо работает с большими наборами данных, но имеет более высокую пространственную сложность, чем быстрая сортировка. Наконец, пирамидальная сортировка — это алгоритм сортировки на месте, который эффективно обрабатывает маленькие наборы данных, но может выдать нелучший результат при работе с большими наборами.

Понимая характеристики различных алгоритмов сортировки, вы сможете выбрать тот, который больше всего подходит для вашего конкретного случая, и оптимизировать производительность своего приложения.

Выбирая лучший алгоритм для проекта, важно учитывать множество факторов, и один из наиболее важных — размер набора данных, поскольку он влияет на эффективность алгоритмов.

Например, для работы с хорошо структурированными данными подойдет один алгоритм, а с неструктурированными — другой. Наконец, важно учитывать конкретный вариант использования. В зависимости от предполагаемого сценария применения вашего проекта одни алгоритмы могут оказаться более эффективными, чем другие. Потратив время на тщательный анализ этих факторов, вы сможете принять обоснованное решение о том, какой алгоритм использовать для достижения наилучших результатов.

6.2. Сортировка выбором

Существует множество алгоритмов сортировки. Один из них — сортировка выбором, особенно полезная для новичков в информатике и алгоритмах, поскольку проста и понятна.

По сути, *сортировка выбором* делит набор данных на две части — отсортированную и неотсортированную. Вначале отсортированная часть пуста, а все данные находятся в неотсортированной. Затем алгоритм выбирает самый маленький (или самый большой, в зависимости от порядка сортировки)

элемент из неотсортированной части и перемещает его в конец отсортированной.

Процесс повторяется до тех пор, пока неотсортированная часть не опустеет, то есть пока все данные не будут отсортированы. Несмотря на простоту алгоритма, не стоит недооценивать его полезность.

Важно отметить, что сортировка выбором может оказаться очень неэффективной при работе с большими наборами данных, поскольку ее временная сложность равна O(n^2). Это означает, что ее производительность существенно снижается по мере увеличения размера данных. Тем не менее этот алгоритм может служить хорошим обучающим образцом, поскольку формирует основу для изучения более сложных алгоритмов сортировки.

Рассмотрим пример реализации этого алгоритма на Python:

```
def selection_sort(arr):
    for i in range(len(arr)):
        # Отыскать наименьший в оставшейся неотсортированной
        # части массива
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j

        # Поменять местами найденный минимальный элемент
        # с первым элементом в неотсортированной части массива
        arr[i], arr[min_index] = arr[min_index], arr[i]

numbers = [64, 34, 25, 12, 22, 11, 90]
selection_sort(numbers)
print("Отсортированный массив: ", numbers)
```

Эта реализация на Python сначала инициализирует `min_index` текущим индексом `i`. Затем, если среди оставшихся элементов массива найдется элемент меньше, чем `arr[min_index]`, то `min_index` обновляется. Далее минимальный элемент, найденный в неотсортированной части массива, меняется местами с первым элементом в неотсортированной. В результате отсортированная часть массива увеличивается, а неотсортированная — сжимается, пока, наконец, не опустеет.

К сожалению, сортировка выбором с точки зрения временной сложности ненамного эффективнее пузырьковой сортировки. Временная сложность этого алгоритма составляет O(n^2) из-за вложенных циклов, что делает его

малопригодным для сортировки больших наборов данных. Однако он обладает свойством минимизировать количество перестановок (не более n перестановок), поэтому в ситуациях, когда операции записи являются дорогостоящими, сортировка выбором может быть рабочим вариантом.

Более того, алгоритм сортировки выбором прост и понятен, поэтому с него хорошо начинать изучение данного типа алгоритмов. Разобравшись в особенностях его работы, можно лучше понять, как работают более сложные алгоритмы сортировки.

Кроме того, сортировка выбором используется в нескольких реальных приложениях. Например, она может пригодиться в ситуациях, когда необходимо отсортировать лишь несколько элементов, или при сортировке списка с небольшим количеством повторяющихся элементов. Ее можно использовать и в качестве промежуточного шага в более сложных алгоритмах, требующих, чтобы массив был частично отсортирован.

Наконец, стоит отметить, что сортировку выбором можно в некоторой степени оптимизировать. Например, выбирая минимальный или максимальный элемент с помощью двоичного поиска, можно уменьшить количество необходимых сравнений. Кроме того, распараллеливание и другие методы могут ускорить алгоритм и сделать его пригодным для работы с большими наборами данных.

В заключение отметим, что сортировка выбором, не будучи самым эффективным алгоритмом для сортировки больших наборов данных, имеет свою область применения и важна для изучения новичками. Если вы понимаете принципы сортировки выбором, то вам будет проще понять более сложные алгоритмы, а также оценить потенциальные возможности применения сортировки выбором в реальных сценариях.

Временная сложность сортировки выбором, как уже упоминалось, равна O(n^2), но стоит отметить и ее пространственную сложность. Последняя равна O(1), то есть требует постоянного объема дополнительного пространства независимо от размера входного списка. Это связано с тем, что сортировка выбором — алгоритм сортировки на месте: он не требует дополнительного пространства, растущего по мере увеличения размера входного массива, и вместо этого сортирует список, манипулируя его элементами на месте и не используя дополнительные структуры данных.

Стоит подчеркнуть, что при выборе алгоритма, который будет применяться в реальном сценарии, важно иметь представление о его временной и пространственной сложностях. Иногда лучше выбрать более быстрый алгоритм,

даже если он использует больше места, но в ряде ситуаций важнее, например, минимизировать потребление памяти.

Однако, как уже отмечалось выше, сортировка выбором обычно применяется потому, что является простой и легкой в изучении, а не из-за своей эффективности. В следующих разделах мы представим другие алгоритмы сортировки с оптимальной временной и пространственной сложностями, подходящие для использования в реальных приложениях.

6.3. Сортировка вставками

Сортировка вставками — еще один простой и вместе с тем фундаментальный алгоритм информатики, который часто используется в приложениях обработки данных. Он относительно прост и сортирует список, добавляя элементы в отсортированный массив по одному, подобно тому как вы сортируете колоду игральных карт.

Прежде чем углубиться в подробности функционирования алгоритма, рассмотрим его работу с помощью простой метафоры. Представьте, что вы играете в карточную игру. В процессе раздачи карт вы помещаете каждую следующую полученную карту в правильное место относительно карт, полученных перед этим. К тому моменту, когда вы получите все карты, все они будут отсортированы по их достоинству!

Рассмотрим еще одну метафору, объясняющую работу алгоритма. Представьте, что вы наводите порядок в своем шкафу. Первоначально перед вами лежит куча одежды, из которой вы выбираете вещи по одной. Беря каждую вещь, вы помещаете ее в правильное место относительно уже отсортированной одежды. После того как вы возьмете последнюю вещь, одежда в шкафу будет рассортирована и разложена по полочкам!

Алгоритм работает аналогично: сортировка массива осуществляется путем вставки каждого следующего элемента в правильное место относительно уже отсортированных элементов. Это простой, но мощный инструмент для эффективной обработки больших объемов данных.

Техническое описание сортировки вставками выглядит так.

1. Первый извлеченный элемент рассматривается как отсортированный подсписок. Вначале этот подсписок содержит только первый элемент исходного списка.

2. Затем из исходного списка извлекается следующий элемент. Если он больше элемента в отсортированном подсписке, то вставляется после него. Если меньше — то перед ним.
3. Этот процесс повторяется, пока не будут перемещены все элементы исходного списка.

Рассмотрим простой пример.

Допустим, у нас есть список `[4,3,2,1]`.

- После первого прохода первое число 4 считается отсортированным подсписком.
- На втором проходе извлекаем число 3. Оно меньше 4, поэтому вставляем его перед 4: `[3,4,2,1]`.
- Следующее число 2 меньше, чем все элементы слева от него, поэтому перемещаем его в начало: `[2,3,4,1]`.
- Наконец, число 1 тоже необходимо переместить вперед: `[1,2,3,4]`.

Вот как можно реализовать сортировку вставками на Python:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Тестирование
print (insertion_sort([4,3,2,1])) # Выведет: [1,2,3,4]
```

С точки зрения эффективности этот алгоритм лучше подходит для сортировки небольших списков. Его временная сложность равна $O(n\^2)$, поэтому он малопригоден для обработки списков большего размера. Однако пространственная сложность алгоритма равна $O(1)$, поэтому он подходит для случаев, когда объем доступной памяти ограничен. Вдобавок алгоритм хорошо работает, когда входной список частично отсортирован, и в лучшем случае требует линейного времени $O(n)$.

Благодаря своей простоте и эффективности сортировка вставками в некоторых ситуациях становится ценным дополнением к уже имеющемуся набору

алгоритмических инструментов. Ее часто используют в качестве базовой составляющей в более сложных алгоритмах.

Однако следует отметить еще одну положительную черту сортировки вставками — это стабильный алгоритм, то есть он сохраняет относительный порядок элементов с одинаковыми ключами сортировки. Иначе говоря, два равных элемента будут помещены в отсортированный массив в том же порядке, в каком они находились во входных данных. Для некоторых приложений, где исходный порядок имеет значение, это может быть важно.

Кроме того, сортировка вставками — потоковый алгоритм: он может сортировать список по мере его получения. В сценариях, где полный список входных данных неизвестен заранее, например при чтении большого файла построчно, такая функция сортировки вставками может быть очень полезна.

На практике, хотя этот алгоритм при работе с большими списками не так эффективен, как более сложные алгоритмы (быстрая сортировка, сортировка слиянием или пирамидальная сортировка), его преимуществом является простота реализации. Более того, он очень хорошо сортирует списки, которые уже почти отсортированы.

6.4. Быстрая сортировка

Быстрая сортировка — широко используемый алгоритм, известный своей эффективностью и выдающейся производительностью. Он назван в честь его способности быстро сортировать большие массивы или списки элементов. Как и сортировка слиянием, быстрая сортировка использует стратегию «разделяй и властвуй», но применяет уникальный подход.

Работа алгоритма начинается с выбора так называемого опорного элемента. Затем алгоритм разделяет другие элементы на два подмассива, помещая в них элементы, которые больше или меньше опорного. Данный этап выбора и разделения на месте требует лишь небольшого дополнительного пространства.

Одной из ключевых особенностей быстрой сортировки является возможность сортировать огромное количество элементов за короткое время, благодаря чему этот алгоритм часто выбирают при выполнении практической работы. Более того, универсальность алгоритма гарантирует возможность его применения к широкому спектру типов данных — от целых чисел до строк. Кроме

того, сортировка на месте и использование стратегии «разделяй и властвуй» делают этот алгоритм ценным инструментом информатики и анализа данных.

Упрощенное описание работы быстрой сортировки выглядит так.

1. **Выбор опорного элемента.** Это один из важных этапов алгоритма. Его можно выбирать случайно или детерминированно на основе определенных критериев, например из середины списка. Опорный элемент служит ориентиром для сравнения с другими значениями в списке и определения того, с какой стороны они должны находиться. Без опорного элемента алгоритм не сможет эффективно сортировать список. Поэтому важно тщательно продумать его выбор и оценить влияние на общую производительность алгоритма. В следующих разделах мы продемонстрируем, как используется опорный элемент и как различные методы его выбора могут влиять на производительность.
2. **Разделение.** Еще один ключевой этап алгоритма. Элементы массива переставляются так, чтобы все элементы меньше опорного переместились влево, а все элементы больше опорного — вправо. Этот процесс оказывает решающее влияние на эффективность алгоритма, поскольку позволяет рекурсивно сортировать подмассивы меньшего размера. Разделяя массив, алгоритм быстрой сортировки способен «разделять и властвовать», что в итоге приводит к полностью отсортированному массиву. Выбор опорного элемента может сильно повлиять на эффективность алгоритма, причем одни методы выбора оказываются более действенными, чем другие.
3. **Рекурсия.** Вышеупомянутые шаги выполняются снова и применяются отдельно к подмассиву элементов с меньшими значениями и к подмассиву с бóльшими значениями. Это позволяет алгоритму проанализировать и отсортировать весь массив, разбивая его на более мелкие подмассивы и анализируя каждый из них отдельно. Такой подход гарантирует точность и эффективность сортировки.

Чтобы лучше понять процесс, рассмотрим пример.

Возьмем следующий массив целых чисел и отсортируем его в порядке возрастания:

```
numbers = [7, 2, 1, 6, 8, 5, 3, 4]
```

Выберем последний элемент `4` в качестве опорного. Цель разделения — переместить числа меньше 4 влево, а числа больше 4 — вправо.

После разделения массив будет выглядеть так:

```
numbers = [2, 1, 3, 4, 8, 5, 7, 6]
```

Затем рекурсивно применим быструю сортировку к двум подмассивам: `[2, 1, 3]` и `[8, 5, 7, 6]`.

Теперь реализуем этот алгоритм на Python:

```
def quick_sort(array):
    if len(array) <= 1:
        return array
    else:
        pivot = array[len(array) // 2]
        less_than_pivot = [x for x in array if x < pivot]
        equal_to_pivot = [x for x in array if x == pivot]
        greater_than_pivot = [x for x in array if x > pivot]
        return quick_sort(less_than_pivot + equal_to_pivot +
               quick_sort(greater_than_pivot)

numbers = [7, 2, 1, 6, 8, 5, 3, 4]
print(quick_sort(numbers))
```

В этой реализации для простоты мы выбираем опорный элемент из середины массива. Затем формируем три списка: один — из элементов меньше опорного (`less_than_pivot`), другой — из элементов, равных ему (`equal_to_pivot`), и третий — из элементов больше его (`greater_than_pivot`). Затем рекурсивно сортируем списки `less_than_pivot` и `greater_than_pivot` и объединяем результат со списком `equal_to_pivot`, чтобы получить окончательный отсортированный массив.

На практике такая сортировка часто работает быстрее других алгоритмов с временной сложностью $O(n \log n)$. Но в худшем случае может иметь временную сложность $O(n^2)$ и работать довольно медленно. Однако, несмотря на это, в большинстве случаев быстрая сортировка по-прежнему считается хорошим алгоритмом для сортировки данных.

Одна из причин этого заключается в том, что внутренний цикл данного алгоритма можно эффективно реализовать во многих компьютерных архитектурах. Кроме того, большинство реальных данных быстрая сортировка обрабатывает значительно лучше, чем другие алгоритмы сортировки с квадратичной сложностью.

Важно отметить, что у быстрой сортировки есть худший сценарий, который довольно плох и может быть спровоцирован намеренно вредоносным

вводом. Поэтому при использовании быстрой сортировки важно соблюдать осторожность и следить за тем, чтобы входные данные не были специально разработаны для запуска худшего сценария.

Вот несколько дополнительных моментов, на которые стоит обратить внимание при использовании данного алгоритма.

- **Выбор опорного элемента.** Эффективность быстрой сортировки можно повысить, тщательно выбрав опорный элемент. В некоторых случаях, если опорным всегда оказывается самый маленький или самый большой элемент, временная сложность может ухудшиться до $O(n^2)$, что нежелательно. Одним из широко используемых методов, позволяющих избежать этой проблемы, является правило «медианы трех».

  В качестве опорного элемента данный метод выбирает из первого, среднего и последнего элементов массива тот, значение которого окажется посередине. Эта стратегия помогает сбалансировать разделы, даже если входные данные целиком или почти отсортированы, что обеспечивает лучшую производительность в целом. Таким образом, правило «медианы трех» позволяет повысить эффективность быстрой сортировки и снизить риск попадания в худший сценарий.

- **Пространственная сложность.** Алгоритм быстрой сортировки сортирует данные на месте, то есть не нуждается в дополнительной памяти для хранения промежуточных данных, как алгоритм сортировки слиянием, но ему требуется дополнительное пространство на стеке для рекурсивных вызовов.

  Такая сортировка работает быстрее многих других алгоритмов сортировки, но не является стабильной и не сохраняет относительный порядок следования равных элементов. Однако нестабильность можно устранить, используя другую версию быстрой сортировки, называемую стабильной быстрой сортировкой. В ней применяется другой алгоритм разделения, поддерживающий относительный порядок равных элементов, но эта версия менее эффективна.

  В целом быстрая сортировка весьма часто используется для сортировки больших наборов данных из-за ее скорости и эффективного использования памяти. Но выбирая ее, важно учитывать допустимость нестабильности сортировки для конкретной задачи.

- **Практическое применение.** Быстрая сортировка — популярный алгоритм, реализованный в стандартных библиотеках многих языков программирования, в том числе C и C++. Это связано с тем, что алгоритм

имеет хорошую пространственную сложность и для выполнения ему не нужно много памяти. Кроме того, он славится своей скоростью, что особенно важно в приложениях реального времени, когда данные должны обрабатываться быстро. Благодаря этому быстрая сортировка становится ценным инструментом для разработчиков приложений, где требуется выполнение операций в режиме реального времени, таких как видеоигры или финансовые системы.

Быстрая сортировка — излюбленный алгоритм программистов, поскольку прост и универсален. Его легко адаптировать для сортировки различных типов данных, таких как числа, строки и даже сложные объекты. Подобная гибкость делает его полезным инструментом для разработчиков, которым необходимо сортировать данные в разных форматах.

Таким образом, быстрая сортировка — мощный алгоритм, широко используемый в сообществе программистов. Благодаря сочетанию хорошей пространственной сложности, высокой скорости и универсальности он прекрасно подходит для быстрой сортировки данных в приложениях реального времени.

- **Оптимизация.** Чтобы увеличить эффективность алгоритма, при обработке небольших подмассивов можно использовать гибридный подход. Он предполагает переключение на более простой алгоритм сортировки, такой как сортировка вставками, когда размер подмассива уменьшается ниже определенного порога, например меньше десяти.

  Используя такое пороговое значение, алгоритм может переключиться на сортировку вставками подмассивов, содержащих меньше десяти элементов, которая, как известно, более эффективно справляется с небольшими массивами. Гибридный подход может помочь сократить общее время работы алгоритма и повысить его эффективность.

- **Рандомизированная быстрая сортировка.** Этот вариант быстрой сортировки позволяет избежать попадания в наихудший сценарий с квадратичным временем выполнения путем случайного выбора опорного элемента. Он устраняет зависимость времени выполнения от характера входной последовательности данных и гарантирует невозможность попадания в наихудший сценарий. Благодаря рандомизации средняя временная сложность алгоритма составляет $O(n \log n)$, что позволяет ему справляться с сортировкой данных на практике.

  Кроме того, рандомизированная быстрая сортировка была тщательно изучена и проанализирована, что привело к разработке нескольких важных вариантов алгоритма. Некоторые из них содержат гибридную быструю

сортировку, использующую сортировку вставками для упорядочения небольших разделов, и быструю сортировку по медиане трех, которая выбирает опорный элемент по трем случайно выбранным элементам. Исследования показали, что в определенных ситуациях эти варианты улучшают производительность рандомизированной быстрой сортировки.

Кроме того, рандомизированная быстрая сортировка широко используется в самых разных приложениях: от сортировки больших наборов данных в базах данных до сортировки элементов в компьютерной графике. Этот алгоритм эффективно обрабатывает большие наборы данных, поэтому широко используется как специалистами по данным, так и программистами.

- **Многозадачность и параллелизм.** Быстрая сортировка поддается распараллеливанию благодаря применяемому принципу «разделяй и властвуй». Помимо шага, выполняющего деление всего массива, последующие рекурсивные шаги тоже можно распараллелить, запуская каждый рекурсивный вызов на отдельном процессоре. Это может значительно повысить производительность, особенно если размер массива велик и в вычислительной среде имеется несколько доступных процессоров.

  Однако важно отметить, что эффективность распараллеливания по-прежнему будет зависеть от характера данных и распределения рабочей нагрузки между процессорами. Более того, существует несколько методов, с помощью которых можно оптимизировать распараллеливание быстрой сортировки и добиться еще большей производительности. Среди таких методов — балансировка нагрузки, планирование задач и секционирование данных.

  Балансировка нагрузки гарантирует равномерное распределение работы между процессорами, а с помощью механизма планирования можно определить порядок выполнения задач. Секционирование предполагает разделение данных на более мелкие подмножества, которые процессоры могут обрабатывать независимо.

- **Приложения.** Многие отдают предпочтение быстрой сортировке из-за ее средней временной сложности $O(n \log n)$, которая выглядит лучше на фоне многих других алгоритмов сортировки. Однако сортировка связанных списков обычно выполняется с помощью сортировки слиянием, поскольку это стабильный алгоритм сортировки, сохраняющий порядок равных элементов во входном массиве.

  Быстрая сортировка, напротив, не является стабильным алгоритмом и может не сохранять исходный порядок равных элементов. Это важно

учитывать, если у вас есть равные элементы и их порядок в отсортированном массиве должен оставаться таким же, каким он был во входном.

Несмотря на этот недостаток, быстрая сортировка по-прежнему широко используется, поскольку выполняется на месте. Напомним: это означает, что для ее выполнения не требуется дополнительная память, кроме исходного входного массива.

Подытожим: выбор алгоритма сортировки, который лучше всего справится с вашей задачей, зависит от множества факторов, таких как объем и характер распределения данных. Чтобы принять обоснованное решение, важно понимать, как работает каждый из алгоритмов. Рассмотрев их сильные и слабые стороны сквозь призму конкретных требований, вы сможете выбрать тот, который лучше всего соответствует вашим потребностям и оптимизирует производительность вашего приложения. Поэтому найдите время, чтобы изучить и протестировать различные алгоритмы и выбрать тот, который подходит именно вам.

6.5. Сортировка слиянием

Сортировка слиянием — весьма эффективный алгоритм, используемый для сортировки больших наборов данных. Алгоритм следует парадигме «разделяй и властвуй»: задачи разбиваются на более мелкие, которые легче решить. В частности, сортировка слиянием делит неотсортированный список чисел на две половины и затем рекурсивно сортирует каждую из них по отдельности. После сортировки двух половин алгоритм объединяет их, чтобы создать отсортированный список.

Одно из ключевых преимуществ сортировки слиянием по сравнению с другими алгоритмами сортировки — ее способность обрабатывать большие наборы данных. Алгоритм делит набор данных на более мелкие, и его легко распараллелить для выполнения на нескольких процессорах или компьютерах. Поэтому он идеально подходит для сортировки больших наборов данных в распределенных системах, хотя и не является самым быстрым.

Еще одно преимущество сортировки слиянием — стабильность. Алгоритм сортировки считается стабильным, если сохраняет исходный относительный порядок равных элементов. Сортировка слиянием — как раз такой алгоритм. Это свойство особенно полезно в сценариях, где важно сохранять порядок равных элементов.

Концептуально сортировка слиянием работает следующим образом.

1. Если длина списка равна 0 или 1, то считается, что он уже отсортирован. В противном случае список делится на два подсписка примерно равного размера.
2. Каждый подсписок рекурсивно сортируется путем повторного применения сортировки слиянием.
3. Два подсписка объединяются в один отсортированный список.

Сортировка слиянием отличается от многих других алгоритмов сортировки не только стабильностью (исходный порядок следования равных элементов сохраняется), но и временной сложностью $O(n \log n)$ во всех случаях: лучшем, худшем и среднем.

Теперь рассмотрим пример реализации сортировки слиянием на Python:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    return merge(merge_sort(left_half, merge_sort(right_half))

def merge(left, right):
    merged = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] <= right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1

    merged += left[left_index:]
    merged += right[right_index:]

    return merged

numbers = [34, 19, 47, 53, 38, 76, 23, 101, 84, 22]
sorted_numbers = merge_sort(numbers)
print(sorted_numbers)
```

Здесь `merge_sort` — основная функция, вызываемая для сортировки массива. Она делит его на две половины, рекурсивно сортирует их и затем объединяет. Функция `merge` отвечает за объединение двух отсортированных массивов в один отсортированный. Она многократно извлекает наименьший из необъединенных элементов во входных массивах.

Несмотря на свою эффективность, сортировка слиянием не является сортировкой на месте, то есть ей необходимо дополнительное пространство в памяти. Это требование может стать проблемой в сценариях, когда объем доступной памяти ограничен, например во встроенных системах или мобильных устройствах. Однако важно отметить, что сортировка слиянием имеет множество преимуществ, благодаря которым прекрасно подходит для сортировки больших наборов данных, особенно не помещающихся в основную память.

Обсудим еще несколько моментов, которые помогут вам лучше понять, что такое сортировка слиянием.

- **Временная и пространственная сложности.** Сортировка слиянием — эффективный алгоритм сортировки, временная сложность которого во всех случаях равна $O(n \log n)$. Поэтому он лучше подходит для обработки больших наборов данных, чем алгоритмы сортировки с квадратичной временной сложностью, такие как пузырьковая сортировка, сортировка выбором или сортировка вставками. Кроме того, этот алгоритм стабилен и может одинаково эффективно обрабатывать неотсортированные, почти отсортированные или полностью отсортированные данные.

  Однако он не лишен недостатков. Его пространственная сложность равна $O(n)$, то есть алгоритму требуется дополнительная память. Если вы работаете в условиях ограниченного объема памяти, то, возможно, вам лучше рассмотреть возможность использования алгоритма сортировки с меньшей пространственной сложностью.

  Напомним, что сортировка слиянием — это алгоритм, следующий парадигме «разделяй и властвуй». Он рекурсивно разбивает входной массив на меньшие подмассивы, из-за чего может быть менее эффективным, чем другие алгоритмы сортировки, при работе с маленькими массивами. В таких случаях, возможно, лучше использовать другие алгоритмы, например сортировку вставками, более эффективно справляющиеся с небольшими массивами.

  В целом можно сказать, что сортировка слиянием — очень эффективный и стабильный алгоритм сортировки, который особенно полезен при

обработке больших наборов данных. Однако он может не подойти в сценариях с ограниченными объемами памяти или очень маленькими массивами.

- **Стабильность.** Сортировка слиянием — стабильный алгоритм. Он сохраняет исходный относительный порядок элементов, имеющих одинаковые ключи (или значения, по которым выполняется сортировка). Это может быть важно в определенных сценариях, когда исходный порядок имеет значение.

  Например, представьте, что вы сортируете список учащихся по именам. Если двое учащихся имеют одинаковые имена, то иногда желательно сохранить их исходный порядок следования в списке после сортировки. В противном случае можно потерять важную информацию об исходном порядке.

  Кроме того, благодаря стабильности сортировка слиянием часто используется для сортировки объектов с несколькими ключами. Например, если у вас есть список имен и фамилий людей, то вы можете отсортировать список сначала по фамилии, а затем по имени. Алгоритм позволяет добиться этого, сохраняя исходный порядок людей в списке.

  Таким образом, стабильность сортировки слиянием — важная особенность, которая может иметь решающее значение в конкретных случаях использования, когда важен исходный порядок элементов. Это свойство также позволяет сортировать объекты с несколькими ключами, сохраняя исходный порядок в списке.

- **Применение.** Сортировку слиянием часто выбирают для сортировки связанных списков, поскольку ей достаточно последовательного доступа. Это означает, что алгоритм может эффективно сортировать связанные списки, которые по своей природе являются последовательными. Сортировка слиянием также с успехом может использоваться для обработки внешних данных, хранящихся на носителях с медленным доступом, например на жестком диске. Линейное чтение и запись с жесткого диска выполняются быстрее, поэтому благодаря сортировке слиянием вы можете получить дополнительное преимущество, последовательно обращаясь к диску и оптимально используя характеристики производительности устройства.

  Кроме того, сортировка слиянием имеет ряд иных преимуществ перед другими алгоритмами сортировки. Например, это стабильный алгоритм сортировки, сохраняющий исходный относительный порядок равных элементов. Это может быть важно в некоторых приложениях, например при сортировке данных, имеющих несколько атрибутов, или при работе с данными, которые уже частично отсортированы.

- **Варианты.** Сортировка слиянием — широко используемый алгоритм сортировки, имеющий несколько вариантов, предназначенных для повышения производительности в конкретных условиях. Один из самых популярных — алгоритм Timsort, который по умолчанию применяется в Python и Java. Для маленьких массивов Timsort использует сортировку вставками, поскольку этот алгоритм более эффективен при работе с маленькими объемами данных. Для обработки больших массивов Timsort применяет сортировку слиянием.

  Другой вариант — сортировка слиянием снизу вверх, использующая итеративный подход вместо рекурсивного и последовательно сортирующая подмассивы увеличивающегося размера, начиная с подмассивов с размером 1. Большие массивы этот вариант часто обрабатывает более эффективно, чем традиционная рекурсивная сортировка слиянием. В целом существует множество способов оптимизации сортировки слиянием для конкретных случаев использования.

- **Несравнительная сортировка.** Алгоритмы сортировки можно разделить на два типа: сравнительные и несравнительные.

  Сравнительные алгоритмы (например, быстрая сортировка) наиболее распространены и в процессе работы сравнивают элементы.

  Несравнительные алгоритмы не сравнивают элементы, чтобы их сортировать, а разбивают задачу на более мелкие части. Сортировка слиянием — пример алгоритма несравнительной сортировки, использующего подход «разделяй и властвуй». Он сортирует мелкие части, а затем объединяет их, чтобы получить окончательный отсортированный список. Это делает сортировку слиянием более эффективной и стабильной, чем другие алгоритмы сортировки.

- **Параллельная обработка.** Сортировка слиянием хорошо подходит для параллельных вычислений. Это связано с принципом «разделяй и властвуй», который позволяет распределять сортировку подмассивов по нескольким потокам или даже машинам. Благодаря этому процесс сортировки больших наборов данных можно значительно ускорить. Кроме того, параллельные вычисления позволяют снизить рабочую нагрузку на одну машину, что способствует более эффективному использованию ресурсов.

  Вдобавок возможность распределять задачи по нескольким потокам или машинам может помочь повысить отказоустойчивость, поскольку любые сбои в конкретном потоке или машине можно изолировать и ограничить их влияние на весь процесс сортировки. В целом благодаря

использованию параллельных вычислений сортировка слиянием может стать еще более мощным инструментом быстрой и эффективной сортировки больших наборов данных.

- **Потоковая сортировка.** Сортировка слиянием требует, чтобы все входные данные были доступны до начала сортировки. Это означает, что данный алгоритм не является потоковым и не может обрабатывать входные данные последовательно, по частям, не имея всей входной информации с самого начала.

  Сортировка слиянием не подходит для ситуаций, когда данные принимаются в потоковом режиме. Однако существуют другие алгоритмы сортировки, которые могут сортировать данные по мере их получения. К ним относятся сортировка вставкой, сортировка выбором и пузырьковая сортировка, которые могут работать с частичными входными данными и использоваться в потоковых сценариях.

- **Адаптивность.** Адаптивные алгоритмы сортировки используют существующий порядок входных данных для повышения эффективности. Они могут завершиться быстрее, если входные данные частично отсортированы, тогда как неадаптивные алгоритмы не обладают этим свойством.

  Сортировка слиянием — неадаптивный алгоритм, не дает преимуществ при работе с частично отсортированными данными и всегда имеет временную сложность $O(n \log n)$. Однако этот недостаток адаптивности компенсируется тем фактом, что сортировка слиянием является стабильным алгоритмом, сохраняющим исходный относительный порядок равных элементов.

  Благодаря этому алгоритм сортировки слиянием подходит для обработки больших наборов данных, особенно когда они хранятся на дисках или в памяти с ограниченной скоростью доступа. Наконец, сортировка слиянием — это алгоритм сортировки на основе сравнения, то есть он полагается только на сравнение пар элементов для определения их относительного порядка. Поэтому сортировка слиянием является универсальным алгоритмом, который можно применять к широкому спектру типов данных, если их можно сравнивать с помощью четко определенного способа упорядочения.

Выбор алгоритма во многом зависит от контекста его применения. Имея глубокое понимание различных алгоритмов, вы сможете принять обоснованное решение, выбирая лучший алгоритм для своих задач. Например, при выборе алгоритма сортировки следует учитывать требования задачи, такие как размер и характер данных, а также ограничения системы.

6.6. Пирамидальная сортировка

Пирамидальная сортировка — весьма эффективный и широко используемый алгоритм, который может сортировать массив или список на месте, используя структуру данных двоичной кучи. Этот алгоритм использует двоичное дерево, известное как двоичная куча, которое поддерживает определенный порядок: *неубывающий (min-heap)* или *невозрастающий (max-heap)*. В куче с неубывающим порядком родительский узел всегда меньше своих дочерних узлов или равен им, тогда как в куче с невозрастающим порядком — всегда больше их или равен им.

Используя этот особый порядок, алгоритм позволяет эффективно сортировать массивы и списки разных размеров. Кроме того, он является универсальным и может быть реализован на различных языках программирования, поэтому может применяться во множестве приложений.

Пирамидальная сортировка делится на два основных этапа.

1. **Формирование кучи.** Алгоритм преобразует неотсортированный входной массив в невозрастающую кучу. Самый большой элемент при этом перемещается в корень дерева. Наибольшее значение гарантированно окажется в верхней части кучи, что упрощает его извлечение при необходимости. Формирование кучи играет решающую роль в подготовке структуры данных к следующему этапу — сортировке. Он гарантирует организацию данных, упрощающую их сортировку и извлечение. Можно сказать, что формирование кучи — базовый процесс, закладывающий основу для применения эффективных алгоритмов сортировки.
2. **Сортировка.** После создания кучи корневой узел (содержащий максимальное значение) меняется местами с последним узлом в куче. Затем размер кучи уменьшается на единицу (в результате последний узел, теперь имеющий максимальное значение, перемещается в отсортированную часть массива), после чего процесс формирования кучи повторяется для оставшихся узлов. Так продолжается до тех пор, пока куча не опустеет, в результате чего получится отсортированный массив.

Рассмотрим пример реализации на Python:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
```

```
    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Этот код создает невозрастающую кучу с помощью функции `heapify`, а затем сортирует массив, меняя местами корень кучи с последним узлом, уменьшая размер кучи на единицу и объединяя в кучу оставшиеся узлы.

Далее перечислены некоторые ключевые характеристики пирамидальной сортировки.

- **Временная сложность.** Пирамидальная сортировка имеет временную сложность $O(n \log n)$ для всех случаев, включая лучший, средний и худший. Это означает, что она одинаково эффективно обрабатывает любые входные данные любого объема. Однако стоит отметить: хотя пирамидальная сортировка и не самый быстрый алгоритм, она подходит в тех ситуациях, когда требуются производительность и стабильность.

  Кроме того, пирамидальная сортировка часто используется в случаях, когда сортируемые данные не хранятся в памяти, а поступают из внешнего источника, например из базы данных или из сети, поскольку при реализации данного алгоритма можно минимизировать количество обращений к памяти и тем самым значительно повысить производительность.

- **Пространственная сложность.** Как вы уже знаете, этот алгоритм выполняется на месте, то есть использует постоянный объем дополнительного пространства, не зависящий от объема данных. Соответственно, его пространственная сложность равна $O(1)$.

  Как уже говорилось, пирамидальная сортировка — сравнительный алгоритм, который сначала организует данные для сортировки в двоичное

дерево или кучу. Затем куча делится на две части: отсортированную и неотсортированную. После этого алгоритм неоднократно меняет местами первый элемент в неотсортированной части и самый большой элемент в отсортированной, перемещая границу между двумя частями на один элемент вправо. Делая это неоднократно, можно отсортировать данные.

В целом можно сказать, что пространственная сложность пирамидальной сортировки равна O(1), а благодаря сортировке на месте этот алгоритм очень полезен в ситуациях, когда объем доступной памяти ограничен.

- **Стабильность.** Пирамидальная сортировка нестабильна, то есть после сортировки равные ключи могут не сохранять первоначальный порядок. В некоторых приложениях отсутствие стабильности, особенно если входные данные содержат много повторяющихся ключей, может иметь серьезные последствия и приводить к непреднамеренным изменениям порядка элементов и неверным результатам или даже к сбоям.

  Поэтому при выборе подходящего метода для конкретной задачи важно учитывать стабильность алгоритмов сортировки. Пирамидальная сортировка может иметь преимущества в виде высокой скорости и эффективного использования памяти. Но чтобы гарантировать надежность и точность конечных результатов, необходимо учитывать недостаточную стабильность этого алгоритма.

- **Сортировка на месте.** Как упоминалось выше, пирамидальная сортировка — это алгоритм сортировки на месте, то есть ему требуется постоянный объем дополнительного пространства, не зависящий от объема сортируемого массива. Этим он отличается от других алгоритмов, таких как сортировка слиянием или быстрая сортировка, которым нужно дополнительное пространство, пропорциональное размеру входного массива.

  Стоит отметить, что алгоритмы сортировки на месте иногда могут оказаться менее эффективными с точки зрения временной сложности, поскольку им приходится выполнять больше перестановок или сравнений, чем алгоритмам, использующим дополнительную память. Тем не менее пирамидальная сортировка часто применяется для сортировки больших наборов данных в средах с ограниченными ресурсами.

Пирамидальная сортировка прекрасно подходит для работы с неотсортированными данными, особенно когда нужен надежный алгоритм, быстро справляющийся с большими наборами данных и обеспечивающий стабильную производительность. Обязательно попрактикуйтесь в его реализации, поскольку, освоив этот алгоритм, вы сможете решить множество задач программирования!

Пирамидальная сортировка показывает отличную временную сложность в среднем случае и не требует много дополнительной памяти, но у нее есть несколько недостатков. Один из наиболее существенных — несколько худшая производительность, чем у других алгоритмов сортировки $O(n \log n)$, таких как быстрая сортировка и сортировка слиянием. Это связано с тем, что внутренний цикл может быть довольно сложным и не использовать преимущества пространственной локальности, поэтому производительность кэша может быть не слишком хорошей. Это означает, что сравниваться могут элементы, далеко отстоящие друг от друга в памяти, в связи с чем работа в большинстве современных систем с кэшем может замедлиться.

Помните, что выбор алгоритма сортировки во многом зависит от особенностей решаемой задачи, таких как объем входных данных, требования к стабильности и объему доступной памяти и других факторов.

6.7. Практические задачи

Пришло время практики. Вам предстоит решить некоторые задачи, чтобы вы могли закрепить ваши знания об алгоритмах сортировки. Не волнуйтесь, если поначалу будете испытывать сложности, — это часть процесса обучения.

Задача 1. Реализация пузырьковой сортировки

Начнем с наиболее простой задачи. Попробуйте реализовать алгоритм пузырьковой сортировки на любом языке программирования по вашему выбору. Кратко напомним, как работает такая сортировка.

Этот алгоритм многократно меняет местами соседние элементы, если они расположены в неправильном порядке. Он продолжает повторять итерации, пока не наступит момент, когда в одной из них не потребуется сделать ни одной перестановки.

Решение

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j +1] = arr[j + 1], arr[j] # Перестановка
    return arr
```

Задача 2. Анализ худшего случая быстрой сортировки

Попробуйте объяснить, почему временная сложность быстрой сортировки может снизиться до $O(n^2)$. При каких условиях это происходит и как этого избежать?

Задача 3. Сортировка слиянием связанного списка

В большинстве примеров сортировки слиянием используются массивы, но этот алгоритм может работать и с другими структурами данных, причем очень эффективно. Попробуйте реализовать сортировку слиянием для связанного списка и объясните, как это обстоятельство меняет пространственную сложность алгоритма.

Решение

Реализация сортировки слиянием для связанного списка будет выглядеть иначе, поскольку связанные списки не поддерживают произвольный доступ. Однако и в этом случае сортировка слиянием может быть очень эффективной благодаря своему последовательному характеру и возможности вставлять узлы в начале списка за постоянное время. Пространственная сложность остается равной $O(n)$, так как в процессе сортировки вам придется создавать новые узлы для хранения отсортированного списка.

Вот реализация сортировки слиянием для связанного списка на Python. Здесь определяется простой класс узла связанного списка со значением и указателем на следующий узел:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

def merge_sort_linked_list(head):
    if head is None or head.next is None:
        return head

    # Поиск середины и разбиение списка
    mid = get_middle(head)
    mid_next = mid.next
    mid.next = None
```

```
    # Рекурсивная сортировка значений
    left = merge_sort_linked_list(head)
    right = merge_sort_linked_list(mid_next)

    # Слияние отсортированных половин
    return merge_sorted_lists(left, right)
```

Задача 4. Формирование кучи на основе массива

Пирамидальная сортировка начинается с преобразования массива в невозрастающую кучу. Попробуйте написать функцию, которая преобразует массив в кучу. Для этого перестройте массив так, чтобы он соответствовал свойству невозрастающей кучи: значение любого данного узла `i` должно быть больше или равно значениям его дочерних элементов.

Решение

Вот реализация функции `heapify` на Python:

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def build_max_heap(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
```

Задача 5. Стабильность сортировки

Вспомните, какие из рассмотренных нами алгоритмов сортировки стабильны, а какие — нет. В каких реальных сценариях важно сохранять исходный относительный порядок равных элементов в отсортированном списке?

Помните, что эти практические задачи нужны не просто для того, чтобы вы могли найти решение, — они помогают вам лучше понять работу алгоритмов сортировки. Поэтому внимательно обдумайте каждый шаг и не стесняйтесь вернуться к тому или иному разделу книги, чтобы освежить информацию в памяти, если что-то покажется вам сложным. Все это — часть пути становления опытного программиста. Успехов!

Резюме

На очередном этапе нашего увлекательного путешествия по миру алгоритмов мы познакомились с алгоритмами сортировки — ключевыми инструментами информатики, которые служат для упорядочения списков элементов по возрастанию или убыванию. Эти алгоритмы очень важны, поскольку весьма широко используются в самых разных областях.

Первым мы рассмотрели алгоритм пузырьковой сортировки — простой и понятный, но малоэффективный алгоритм с временной сложностью $O(n$^2) в худшем случае. Он многократно переставляет местами соседние элементы, расположенные в неправильном порядке, в результате чего более крупные элементы «всплывают» на свои правильные позиции.

Затем мы перешли к сортировке выбором — еще одному алгоритму с временной сложностью $O(n$^2). Он работает немного иначе: неоднократно выбирает самый маленький (или самый большой) элемент из неотсортированной части списка и перемещает его в начало, формируя отсортированную часть.

Далее мы исследовали сортировку вставками. Этот алгоритм строит отсортированный список по одному элементу за раз, подобно тому как мы упорядочиваем игральные карты: берем по одной и кладем в правильное место. Этот алгоритм эффективно справляется с небольшими или частично отсортированными списками, но имеет ту же временную сложность в наихудшем случае, что и предыдущие два, то есть $O(n$^2).

Затем мы перешли к более сложным алгоритмам. Быстрая сортировка использует парадигму «разделяй и властвуй», деля список по опорному элементу и рекурсивно сортируя подсписки. Временная сложность этого алгоритма в худшем случае равна $O(n$^2), однако на практике он работает намного лучше и используется часто из-за своей средней временной сложности $O(n \log n)$.

Сортировка слиянием — еще один алгоритм, использующий парадигму «разделяй и властвуй». Он разбивает список на две половины, сортирует их по отдельности, а затем объединяет в общий отсортированный список. Этот алгоритм гарантирует временную сложность $O(n \log n)$ во всех сценариях.

И наконец, мы рассмотрели алгоритм пирамидальной сортировки. Он основан на сравнении и использует структуру данных двоичной кучи. Сначала из входных данных он создает невозрастающую кучу, затем меняет местами максимальный и последний элементы и снова формирует кучу из оставшихся данных. Его временная сложность тоже во всех случаях равна $O(n \log n)$.

В конце главы мы предложили вам решить несколько практических задач, чтобы вы могли закрепить ваши знания об алгоритмах сортировки, попрактиковаться в их реализации, проанализировать их поведение и выявить различия.

Имея представление об этих базовых алгоритмах, их временной и пространственной сложности, а также пригодности к использованию в разных сценариях, вы сможете эффективно применять их в вашей работе. Они послужат вам хорошей отправной точкой, с которой можно начать изучение многих других алгоритмов сортировки, и помогут сформировать четкое понимание основных принципов сортировки в информатике. Продолжайте практиковаться и исследовать — и обретете новые навыки и идеи.

Глава 7
ГРАФОВЫЕ АЛГОРИТМЫ

Графы находят широкое применение в разных областях: от социальных сетей до систем GPS. Графовые алгоритмы позволяют извлекать информацию из этих сложных структур. Анализируя графы, можно обнаружить скрытые закономерности и взаимосвязи, которые не сразу бросаются в глаза. В этой главе мы рассмотрим некоторые из наиболее важных графовых алгоритмов, в том числе поиск в ширину, поиск в глубину и алгоритм Дейкстры.

Кроме того, мы рассмотрим некоторые примеры практического применения графовых алгоритмов, такие как планирование маршрутов, сетевой анализ и системы рекомендаций. Вы увидите, насколько эффективны и универсальны эти алгоритмы.

7.1. Введение в теорию графов

В этом разделе мы углубимся в основы теории графов — увлекательного раздела математики, существующего с XVIII века. Теория графов — область исследований, в которой основное внимание уделяется *графам*, представляющим собой набор объектов (вершин или узлов), соединенных *связями* (ребрами или дугами). Графы можно использовать для моделирования широкого спектра реальных систем, в том числе компьютерных и социальных сетей и даже биологических систем.

Говоря о графах, мы на самом деле имеем в виду два основных компонента: вершины (или узлы) и ребра. Каждая *вершина* — это объект, а каждое *ребро* — связь между парой объектов. Например, рассмотрим группу друзей. Каждого друга можно представить в виде вершины, и если два друга знакомы между собой, то мы можем добавить ребро между соответствующими вершинами. Тот же принцип применим ко множеству реальных систем: от транспортных до социальных сетей.

Таким образом, можно сказать, что теория графов позволяет моделировать и анализировать сложные системы визуальным и интуитивно понятным

способом. Понимая основы данной теории, мы можем начать изучать многочисленные области применения графов и информацию, которую они могут нам дать.

Вот простое представление графа в Python с использованием словарей:

```
grapf = {
    "Alice": ["Bob", "Charles"],
    "Bob": ["Alice", "David"],
    "Charles": ["Alice", "David"],
    "David": ["Bob", "Charles"]
}
```

В этом графе вершинами являются `"Alice"`, `"Bob"`, `"Charles"` и `"David"`. Ребра представлены связями в списках. Например, вершина `"Alice"` связана с вершинами `"Bob"` и `"Charles"`.

Существуют два основных типа графов: ориентированные и неориентированные. С одной стороны, в неориентированном графе связи между вершинами двунаправленные, то есть можно переходить по ребрам в любом направлении. Например, если Алиса (Alice) дружит с Бобом (Bob), то Боб, в свою очередь, дружит с Алисой. С другой стороны, ориентированный граф (или орграф) имеет ребра с заданным направлением. Например, если Алиса отправляет электронное письмо Бобу, это не означает, что и он отправляет ей письмо.

Кроме того, графы могут быть взвешенными и невзвешенными. Во взвешенном графе каждое ребро имеет определенное значение или вес. Эти веса (в зависимости от решаемой задачи) могут обозначать расстояние, затраты, приоритет или любую другую метрику, количественно определяющую связь между двумя вершинами.

Усвоение этих фундаментальных понятий служит ключом к пониманию более сложных алгоритмов и идей теории графов, которые мы рассмотрим далее в этой главе. Не торопитесь. Если вам понадобится освежить в памяти некие знания — сделайте это, не стесняясь! Помните: путь обучения — это не гонка, а путешествие, полное открытий.

Вы можете спросить себя: «Почему теория графов так важна?» или «Где она применяется?» Теория графов широко используется в различных областях, таких как информатика, физика, химия, биология и социальные науки, и это лишь некоторые примеры.

Например, рассмотрим Интернет. Это обширная сеть взаимосвязанных устройств. Каждое из них (будь то компьютер, сервер или смартфон) можно представить в виде вершины, а соединение между ними — как ребро. Таким образом, весь Интернет можно смоделировать как огромный граф! Благодаря пониманию структуры этого графа можно оптимизировать маршрутизацию потоков данных, повысить безопасность и управлять многими другими аспектами Сети.

В анализе социальных сетей пользователи представлены как вершины, а их отношения — как ребра. Анализируя этот граф, можно выявить структуру сообщества, влиятельных пользователей и другие социальные динамики.

В биологических исследованиях теория графов используется для генетического картирования организмов. Вершины представляют гены, а ребра — связи между ними, что позволяет исследователям получать бесценную информацию о структуре и функциях генов.

Таким образом, теория графов невероятно важна. Частота ее реального применения говорит о том, что эффективные алгоритмы анализа графов, которые мы будем изучать в следующих разделах, необходимы.

Наконец, графы могут быть циклическими или ациклическими. В циклическом графе есть хотя бы один путь, который приведет вас к начальной вершине, если следовать по ребрам в том направлении, которое они указывают. В ациклическом графе таких путей нет.

Каждый из этих типов графов представляет уникальные задачи, требующие разных стратегий и алгоритмов для их решения. Самое интересное в теории графов — это разнообразие задач, которые она может представлять и решать. Далее в этой главе и следующей мы будем изучать различные типы графов, а также алгоритмы и стратегии, используемые для их анализа.

7.2. Поиск в глубину

Поиск в глубину (Depth-First Search, DFS) — метод, используемый для обхода или поиска в дереве или графе. Алгоритм начинает работу с корневого узла и обходит граф как можно дальше вдоль каждой ветви, прежде чем вернуться назад. Этот процесс продолжается до тех пор, пока все узлы не будут исследованы.

Проиллюстрировать эту идею можно с помощью такого примера: представьте, что вы находитесь в лабиринте. Если вы решите использовать поиск в глубину, то вам нужно выбрать одно направление и двигаться в эту сторону настолько далеко, насколько это будет возможно. Зайдя в тупик, вы должны вернуться и выбрать новое направление. Этот процесс будет повторяться до тех пор, пока вы не изучите все возможные пути.

Алгоритм DFS обычно используется в различных приложениях, таких как определение связности графа, поиск пути между двумя узлами и выявление циклов в графе. Этот простой и мощный алгоритм позволяет решать самые разные задачи, связанные с теорией графов и структурами данных.

Вот более подробное объяснение поиска в глубину.

1. **Инициализация.** Сначала выбирается узел графа, который будет считаться начальным, и помечается как посещенный. Это важный первый шаг, поскольку здесь закладывается основа для остальной части алгоритма обхода графа. Нужно тщательно продумать, какой узел выбрать в качестве начального, так как это может сильно повлиять на общую эффективность обхода. Выбрав узел и отметив его как посещенный, вы можете начать процесс обхода графа.
2. **Исследование.** Одним из важных аспектов текущего узла является его граница, состоящая из всех узлов, до которых можно добраться непосредственно из него. Изучая эту границу, можно лучше понять возможные пути дальнейшего движения и потенциальные результаты, которые могут возникнуть. Более того, изучая каждый из пограничных узлов и анализируя содержащуюся в них информацию, мы можем принимать более обоснованные решения о выборе направления дальнейшего движения и предпринимаемых действий. Этот процесс исследования и анализа необходим для достижения успеха в любом начинании и является ключевой составляющей принятия эффективных решений.
3. **Рекурсия.** Это процесс, следуя которому функция вызывает себя снова и снова, пока не выполнится определенное условие. В контексте алгоритма поиска в глубину для каждого узла на границе алгоритм проверяет, был ли он посещен. Если нет, то алгоритм рекурсивно применит к нему поиск в глубину, то есть будет вызывать функцию снова и снова, пока не посетит все узлы и условие не будет выполнено. Эта важная концепция информатики и программирования используется во многих алгоритмах для эффективного решения сложных задач.

Рассмотрим реализацию поиска в глубину на Python:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

visited = [] # Список посещенных узлов

def dfs(visited, graph, node):
    if node not in visited:
        print(node)
        visited.append(node)
        for neighbor in graph[node]:
            dfs(visited, graph, neighbor)

# Тест
dfs(visited, graph, 'A')
```

В этой программе на Python начальным является узел `'A'`. Из `'A'` алгоритм сначала переходит в `'B'`, затем в `'D'`, поскольку `'B'` и `'D'` — первые узлы в соответствующих списках. Достигнув узла `'D'`, не имеющего соседей, алгоритм возвращается в узел `'B'` и посещает узел `'E'`, а затем `'F'`. Наконец, он возвращается в `'A'` и посещает узел `'C'` и его соседа `'F'`. Однако `'F'` уже помечен как посещенный, поэтому DFS завершает работу.

Таким образом, прежде чем вернуться, алгоритм исследует каждую ветвь графа, насколько это возможно. Благодаря такому подходу DFS эффективно решает определенные задачи, такие как поиск связанных компонентов или поиск пути от одного узла к другому. Кроме того, алгоритм справляется и с топологической сортировкой графа. Однако для таких задач, как поиск кратчайшего пути, DFS может не подойти и более эффективным может оказаться поиск в ширину или алгоритм Дейкстры.

Важно помнить, что алгоритмы подобны инструментам: для разных задач нужны разные инструменты. Выбор правильного алгоритма может существенно повлиять на эффективность программы или системы. DFS подходит не для всех задач, а для определенных, и этот ценный инструмент должен иметь в своем арсенале каждый программист.

Мы хотели бы углубиться в один из фундаментальных приемов, связанных с DFS, — возврат. Этот алгоритмический подход широко используется для

решения вычислительных задач, требующих выполнения последовательности шагов или перебора вариантов.

Алгоритм с возвратом перебирает все возможные варианты или шаги в поисках решения. Если один вариант не дает желаемого результата, то алгоритм отступает назад (возвращается) и выбирает следующий вариант. Этот метод очень часто используется для решения головоломок, таких как судоку или поиск путей через лабиринт.

Чтобы проиллюстрировать эту концепцию, рассмотрим пример использования DFS с возвратом для поиска пути через простой лабиринт. Вообразите двумерную сетку, представляющую лабиринт, где единицы — стены, а нули — открытые проходы. Цель — перейти из верхнего левого угла в нижний правый. Алгоритм DFS с возвратом исследует все возможные пути и выполняет возврат всякий раз, когда заходит в тупик. Это позволяет алгоритму находить оптимальный путь через лабиринт.

Вот как этот алгоритм можно реализовать на Python:

```
def solveMaze(maze):
    # Создать матрицу решений того же размера, что и лабиринт
    solution = [[0 for _ in range(len(maze[0]))] for _ in range(len(maze))]

    # Функция поиска прохода в лабиринте, использующая алгоритм с возвратом
    def solveMazeUtil(maze, x, y, solution):
        # если ячейка x, y - это выход из лабиринта, вернуть True
        if x == len(maze) - 1 and y == len(maze[0]) - 1:
            solution[x][y] = 1
            return True

        # Проверить, является ли ячейка maze[x][y] проходом
        if x >= 0 and x < len(maze) and y >= 0 and y < len(maze[0])
          and maze[x][y] == 0:
            # Пометить ячейку x, y как часть пути решения
            solution([x][y] = 1

            # Сделать шаг в направлении x
            if solveMazeUtil(maze, x + 1, y, solution):
                return True

            # Если шаг в направлении x не приводит к решению,
            # то сделать шаг в направлении y
            if solveMazeUtil(maze, x, y + 1, solution):
                return True

            # Если ни один из предыдущих шагов не приближает к решению, то
            # ВОЗВРАТ: снять метку с ячейки x, y как части пути решения
            solution([x][y] = 0
            return False
```

```
        return False

    # Если путь найден, то решение будет содержать этот путь
    # Иначе считать, что решения не существует
    if solveMazeUtil(maze, 0, 0, solution) is False:
        print("No path exists")
        return False

    # Вывести путь
    for i in solution:
        print(i)
    return True

# Тест
maze = [[0, 1, 0, 1, 1],
        [0, 0, 0, 0, 0],
        [1, 1, 1, 1, 0],
        [1, 1, 1, 1, 0],
        [1, 1, 1, 0, 0]]

solveMaze(maze)
```

Данная программа выведет путь в лабиринте от первой ячейки к последней.

Это один из примеров, как DFS можно использовать для решения сложных задач с помощью возвратов. Надеюсь, это поможет вам глубже понять тему.

7.3. Поиск в ширину

Поиск в ширину (Breadth-First Search, BFS) — это алгоритм обхода графа, который посещает узлы поуровнево, начиная с корневого и двигаясь вширь. То есть сначала он посещает все узлы, ближайшие к корню, и только потом переходит на следующий уровень.

BFS — эффективный алгоритм поиска кратчайшего пути между двумя узлами графа или дерева. В процессе поиска он исследует все возможные пути от начального узла до конечного и гарантирует, что найденный путь будет кратчайшим.

Алгоритм используется не только для поиска кратчайших путей, но и для обхода двоичных деревьев и особенно полезен для поиска ближайших соседей определенного узла в графе или дереве, поскольку посещает узлы поуровнево и гарантирует обход в первую очередь ближайших соседей.

Как следствие, BFS является важным алгоритмом информатики и широко используется в различных приложениях, например в сетевой маршрутизации, интеллектуальном анализе данных и обработке изображений.

Алгоритм работает следующим образом.

1. Выбирается начальный узел (вершина).
2. Исследуются все вершины, прилегающие к начальному узлу.
3. Выполняется переход на следующий уровень, и исследуются все вершины там.
4. Процесс повторяется, пока не будут исследованы все вершины.

Рассмотрим работу алгоритма предметно на простом примере псевдокода:

```
function BFS(graph, root){
    create an empty queue Q
    create an array visited of size = number of nodes in the graph,
      and set all to False

    enqueue root into Q
    set visited[root] = True

    while Q is not empty {
        current_node = dequeue from Q
        print current_node

        for each node i adjacent to current_node {
            if visited[i] is False {
                enqueue i into Q
                set visited[i] = True
            }
        }
    }
}
```

Функция `BFS` поддерживает очередь посещенных узлов. Первым делом она посещает корневой (начальный) узел, затем всех его соседей, затем соседей соседей и т. д.

Вот пример реализации `BFS` на Python для обхода простого невзвешенного графа, представленного списком смежности:

```
from collections import deque

def BFS(graph, root):
    visited = [False] * (len(graph))
    queue = deque()
```

```
    queue.append(root)
    visited[root] = True

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")

        for neighbor in graph[vertex]:
            if visited[neighbor] == False:
                queue.append(neighbor)
                visited[neighbor] = True

# определение графа
graph = {
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
}

BFS(graph, 2) # Выведет: 2 0 3 1
```

Этот код начинает обход графа с узла 2 и посещает сначала каждый узел, примыкающий к нему, затем узлы, соседние с ними, и т. д., пока не будут посещены все узлы.

Поиск в ширину — мощный алгоритм, которым должен овладеть каждый, кто изучает информатику. Он широко применяется на практике и считается идеальной стратегией решения многих задач, связанных с графами, таких как анализ социальных сетей, веб-сканирование, сетевое вещание и многие другие. Поэтому начните практиковать BFS сегодня и раскройте весь его потенциал!

7.3.1. Временная сложность BFS

В худшем сценарии BFS исследует все вершины и ребра, соответственно, имеет временную сложность O(V + E), где V — количество вершин, а E — количество ребер. Однако эту сложность можно улучшить с помощью различных методов оптимизации, таких как использование списка смежности вместо матрицы смежности.

С точки зрения пространственной сложности BFS требует объем памяти O(V), поскольку в худшем случае ему придется поместить в очередь все вершины графа. Это может быть недостатком в случае больших графов, но, опять же, существуют стратегии оптимизации использования памяти.

В целом BFS — универсальный и эффективный алгоритм исследования графов, а его временную и пространственную сложности можно улучшить с помощью некоторых методов оптимизации.

Алгоритм известен прежде всего способностью обходить все связанные вершины в графе. Но мы обсудим и некоторые другие важные случаи его применения.

1. **Поиск кратчайшего пути и планирование.** Поиск в ширину обычно используется для поиска кратчайшего пути в невзвешенном графе и особенно полезен при планировании процессов. Он позволяет оптимизировать порядок выполнения задач и минимизировать время, необходимое для их выполнения.

   Анализируя взаимосвязи между задачами и их зависимостями, алгоритм может помочь определить наиболее эффективный путь достижения желаемого результата. BFS универсален, поэтому применяется в широком спектре алгоритмов планирования и является базовым инструментом различных областей, таких как информатика, логистика и управление проектами.

2. **Обнаружение циклов в неориентированном графе.** BFS можно использовать для выявления циклов в графе. Это важная задача теории графов. Говорят, что граф содержит цикл, если существует путь, который начинается и заканчивается на одной и той же вершине.

3. **Поисковые роботы.** Поисковые системы используют веб-роботов для индексирования веб-страниц, и BFS — один из алгоритмов, который может помочь сориентироваться на обширных просторах Интернета. Сначала алгоритм просматривает начальную страницу и переходит по всем имеющимся ссылкам на другие страницы и т. д. Так он выполняет обход сети Интернет.

   Используя BFS, поисковые системы могут получать информацию из большого количества страниц и создавать свой индекс, который затем можно использовать для предоставления пользователям релевантных результатов поиска. Более того, с помощью веб-роботов поисковые системы поддерживают свой индекс в актуальном состоянии, поскольку могут обнаруживать изменения на веб-страницах и соответствующим образом обновлять индекс. Проще говоря, без веб-роботов поисковые системы не смогут предоставлять пользователям точные и актуальные результаты поиска.

4. **Вещание в сети.** Широковещательные пакеты в сети используют алгоритм BFS, чтобы достичь всех узлов. При выполнении широковещательной передачи каждый пакет отправляется сразу нескольким узлам. Этот процесс

известен как широковещательная рассылка и может быть очень полезным, когда требуется отправить информацию множеству различных устройств.

Алгоритм сначала исследует все соседние узлы, а затем переходит к следующему уровню. Этот метод гарантирует доставку пакета и содержащейся в нем информации всем узлам сети, которые в этом заинтересованы.

5. **GPS-навигация.** Алгоритм BFS позволяет быстро и эффективно найти объекты, находящиеся на заданном расстоянии от определенной начальной точки. Это его свойство с успехом используется в GPS-навигации, например, когда нужно найти близлежащие заправочные станции, рестораны или туристические достопримечательности.

   Кроме того, BFS можно настроить с учетом различных факторов, таких как интенсивность движения или перекрытие дорог, чтобы получить максимально точную и актуальную информацию. В целом BFS — незаменимый инструмент для всех, кто хочет уверенно и легко перемещаться по незнакомой местности.

7.4. Алгоритм Дейкстры

В теории графов широко используется *алгоритм Дейкстры*. Он назван в честь его изобретателя Эдсгера Дейкстры (Edsger Dijkstra). Алгоритм отыскивает кратчайшие пути от исходной вершины ко всем остальным вершинам в графе и действует особенно эффективно, если граф имеет взвешенные ребра. Этот алгоритм может пригодиться в ситуациях, когда граф имеет большое количество вершин и ребер, позволяя максимально эффективно определять кратчайшие пути между узлами.

BFS (поиск в ширину) тоже может найти кратчайший путь в графе, но предназначен для работы с невзвешенными графами, тогда как алгоритм Дейкстры специально разрабатывался для обработки графов с взвешенными ребрами, в которых не все ребра имеют одинаковый вес. Он учитывает вес каждого ребра при определении кратчайшего пути и гарантирует, что найденный путь действительно будет наиболее эффективным.

Помимо практического применения, алгоритм Дейкстры также оказал значительное влияние на информатику. Он подготовил почву для разработки других важных алгоритмов, таких как алгоритм A*, и сыграл ключевую роль в развитии теории графов как области исследований. Поэтому он остается ценным инструментом для исследователей, программистов и всех, кто интересуется этой увлекательной областью математики и информатики.

Вот как работает алгоритм Дейкстры.

1. Поиск начинается с начальной вершины. Расстояние до нее устанавливается равным 0, а расстояние до всех остальных вершин — равным бесконечности (или очень большой величине).
2. Затем создается приоритетная очередь, и в нее добавляется начальная вершина.
3. Пока приоритетная очередь не опустеет:
   - отыскивается вершина с наименьшим расстоянием; пусть это будет вершина U;
   - для каждой вершины V, соседней с U, если расстояние от V до U меньше расстояния до текущей рассматриваемой вершины V, обновляется расстояние до V.
4. Теперь алгоритм имеет кратчайший путь от начальной вершины до каждой имеющейся в графе.

Посмотрим, как выглядит реализация этого алгоритма на Python:

```
import heapq

def dijkstra(graph, start, start_vertex):
    D = {v:float('infinity') for v in graph}
    D[start_vertex] = 0

    priority_queue = [(0, start_vertex)]
    while len(priority_queue) > 0:
        (dist, current_vertex) = heapq.heappop(priority_queue)
        for neighbor, neighbor_distance in graph[current_vertex].items():
            old_distance = D[neighbor]
            new_distance = D[current_vertex] + neighbor_distance
            if new_distance < old_distance:
                D[neighbor] = new_distance
                heapq.heappush(priority_queue, (new_distance, neighbor))
    return D

# Пример графа
graph = {
    'A': {'B':1, 'C':3},
    'B': {'A':1, 'D':3, 'E':6},
    'C': {'A':3},
    'D': {'B':3},
    'E': {'B':6, 'F':9},
    'F': {'E':9}
}

print(dijkstra(graph, 'A')
```

Эта программа выведет кратчайшие расстояния от вершины A до всех остальных вершин.

Хотим еще раз подчеркнуть, что этот алгоритм играет важнейшую роль в теории графов. Поэтому каждому желающему усвоить эту теорию совершенно необходимо иметь полное понимание данного алгоритма. Мы надеемся, что его концептуальное описание и практическая реализация на Python, приведенные выше, помогут вам освоить этот важный инструмент.

Обсудим еще несколько важных моментов.

7.4.1. Взвешенные и невзвешенные графы

Помните, что основные преимущества алгоритма Дейкстры проявляются при работе с взвешенными графами. Если все ребра графа имеют одинаковый вес (или вообще его не имеют), то с неменьшим успехом можно использовать более простые методы, такие как поиск в ширину.

Занимаясь анализом графов, важно различать взвешенные и невзвешенные графы. Во взвешенных графах, в отличие от невзвешенных, каждое ребро имеет вес или стоимость. Алгоритм Дейкстры весьма эффективно анализирует взвешенные графы, поскольку учитывает вес каждого ребра, стремясь найти кратчайший путь между двумя узлами.

Однако при работе с невзвешенным графом алгоритм Дейкстры может оказаться не самым эффективным. Поэтому важно учитывать природу графа и выбирать подходящий алгоритм.

7.4.2. Неотрицательные веса

Алгоритм Дейкстры — популярный алгоритм поиска кратчайшего пути между двумя узлами графа. Однако у него есть ограничение: предполагается, что все веса неотрицательны. По этой причине алгоритм Дейкстры может работать не так, как ожидалось, и давать неверные результаты, если граф содержит ребра с отрицательными весами.

Решить эту проблему могут другие алгоритмы, такие как алгоритм Беллмана — Форда или алгоритм Джонсона, способные обрабатывать графы, имеющие ребра с отрицательными весами. Эти алгоритмы отыскивают

кратчайший путь даже в таких графах. В отличие от алгоритма Дейкстры, они учитывают возможность появления отрицательных весов и соответствующим образом корректируют свои расчеты. Поэтому важно выбрать правильный алгоритм, исходя из особенностей конкретного графа.

7.4.3. Приложения

Алгоритм Дейкстры — универсальный алгоритм, широко используемый в различных областях. Один из наиболее распространенных случаев применения — в протоколах сетевой маршрутизации, таких как OSPF (Open Shortest Path First) и IS-IS (Intermediate System to Intermediate System), где он помогает определить кратчайший путь между двумя сетевыми узлами. Кое-где его используют для уменьшения заторов на дорогах и оптимизации транспортного потока путем поиска кратчайшего пути для каждого транспортного средства.

Более того, алгоритм Дейкстры применяется для оптимизации передачи пакетов между узлами сети, маршрутов движения транспортных средств и людей, например, в системах общественного транспорта. В социальных сетях он используется для предложения новых знакомств, отыскивая кратчайшие пути между пользователями на основе их общих интересов или связей.

Проще говоря, алгоритм Дейкстры играет важную роль в различных областях, предлагая эффективный способ поиска кратчайшего пути между двумя узлами в сети.

7.4.4. Варианты

Как мы уже сказали, алгоритм Дейкстры широко используется в информатике для поиска путей и обхода графов. Но, несмотря на все достоинства классического алгоритма Дейкстры, исследователи разработали несколько вариантов и оптимизаций, стремясь улучшить его.

Одна из таких оптимизаций — алгоритм A*, расширяющий алгоритм Дейкстры и активно применяющийся для поиска путей в графах. Он использует эвристическую функцию и с ее помощью определяет, какие узлы исследовать в первую очередь, вследствие чего может значительно повыситься эффективность алгоритма и уменьшиться количество исследуемых узлов. Более подробно об алгоритме A* мы поговорим чуть позже.

Другой вариант алгоритма Дейкстры — алгоритм двунаправленного поиска, который исследует граф как из начального, так и из конечного узла одновременно, сокращая пространство поиска и улучшая общую производительность алгоритма.

Эти варианты и оптимизации алгоритма Дейкстры еще больше расширили его возможности по решению сложных задач в области информатики.

7.4.5. Визуализация алгоритма Дейкстры

Наблюдение за работой алгоритма поможет вам изучить его более подробно. Обращайтесь к онлайн-ресурсам, предлагающим детальную пошаговую визуализацию его работы. Эти ресурсы помогут вам получить более полное представление о том, как он отыскивает кратчайший путь между двумя узлами в графе.

Кроме того, наблюдая за действиями алгоритма, вы сможете выявить те области, в которых, возможно, испытываете трудности в понимании его работы, и найти любые потенциальные ошибки или ошибки в вашей реализации алгоритма.

Используя эти онлайн-ресурсы, вы сможете получить более полное и точное представление об алгоритме, что в конечном счете поможет вам в будущих исследованиях и практической работе.

7.4.6. Временная сложность

Алгоритм Дейкстры — один из самых популярных алгоритмов поиска кратчайшего пути между двумя узлами графа. Он известен своей эффективностью и скоростью, особенно если реализуется с помощью двоичной кучи или приоритетной очереди, что позволяет добиться временной сложности $O((V + E) \log V)$ и получить более высокую скорость обработки графов с большим количеством вершин и ребер, чем могут дать другие алгоритмы. (V представляет количество вершин, а E — количество ребер.) Да, этот алгоритм подходит не для всех типов графов, но он надежен и широко используется в информатике.

Алгоритм активно применяется на практике, например в системах GPS-навигации для поиска кратчайшего маршрута между двумя точками. В целом

эффективность и универсальность этого алгоритма делают его ценным инструментом для решения широкого круга задач, связанных с графами и анализом сетей.

Помните: овладение алгоритмом Дейкстры, как и любым другим, приходит с практикой. Чем больше вы его используете, тем лучше понимаете и тем эффективнее будут ваши реализации.

7.5. Алгоритм поиска A*

*Алгоритм поиска A** – это высокоэффективный подход к поиску кратчайшего пути между двумя узлами графа. Он сочетает в себе сильные стороны алгоритма Дейкстры и жадного поиска по критерию первого наилучшего совпадения, поэтому применяется в различных приложениях как универсальный инструмент.

Используя эвристическую информацию для управления поиском, алгоритм поиска A* способен принимать обоснованные решения о выборе путей для исследования, что значительно ускоряет процесс поиска. Этот алгоритм получил широкое распространение в таких областях, как информатика, экономика и транспорт, и играет важнейшую роль при решении сложных задач.

Кроме того, алгоритм поиска A* обладает широкими возможностями настройки, вследствие чего пользователи могут настраивать его в соответствии с конкретными потребностями. В целом это мощный и гибкий инструмент, который произвел революцию в подходах к решению задач поиска путей.

Секрет алгоритма A* заключается в использовании функции стоимости, обычно обозначаемой как $f(n)$, которая представляет собой сумму двух других функций:

1) $g(n)$ – точная стоимость пути от начальной точки до текущего узла n;
2) $h(n)$ – предполагаемая стоимость от текущего узла n до целевого узла, обычно рассчитываемая с помощью эвристики (подробнее об этом ниже).

Следовательно, $f(n) = g(n) + h(n)$.

7.5.1. Эвристика в A*

Эвристическая функция $h(n)$ — важный инструмент определения направления к цели в алгоритмах поиска. Она позволяет исследовать наиболее перспективные соседние узлы и быстрее достигать цели. Чаще всего в алгоритме A* используется эвристическая функция с названием «манхэттенское расстояние». Эта эвристика вычисляет расстояние между двумя точками как сумму абсолютных разностей их координат. Например, в сценарии, где начальная точка находится в координатах (1, 2), а целевая точка — в координатах (4, 6), манхэттенское расстояние будет вычислено как $|1 - 4| + |2 - 6| = 3 + 4 = 7$.

Однако важно отметить, что существуют и другие эвристики, которые можно применять в зависимости от задачи. Евклидово расстояние — пример еще одной часто используемой эвристической функции. Она вычисляет расстояние по прямой между двумя точками. В ходе поиска функция дает оценку расстояния между текущим узлом и целью, которая затем используется для определения следующего узла, подлежащего исследованию, что повышает эффективность алгоритма поиска.

Рассмотрим пример алгоритма поиска A*.

Представьте, что мы используем данный алгоритм, чтобы найти кратчайший путь в игре от начальной точки до цели. Нашим графом будет карта игрового поля, а узлы — потенциальными позициями на этой карте. Пусть наша карта выглядит так:

```
S . . . . . G
. # # # . . .
. . . . # . .
```

где:

- `S` — начальная точка;
- `G` — целевое местоположение;
- `.` — открытое пространство;
- `#` — препятствие.

Ниже представлена простая реализация алгоритма с помощью Python. Обратите внимание, что это очень упрощенный пример и реальный код A* будет более сложным, особенно с учетом того, как будет выбираться следующий узел для исследования.

```
import heapq

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_search(graph, start, goal):
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far(start] = 0

    while frontier:
        current = heapq.heappop( frontier) [1]

        if current == goal:
            break

        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(goal, next)
                heapq.heappush(frontier, (priority, next))
                came_from{[next] = current

    return came_from, cost_so_far

# предполагается, что граф (graph), начальная (start) и конечная (goal)
# точки заданы верно
came_from, cost_so_far = a_star_search(graph, start, goal)
```

7.5.2. Временная и пространственная сложности

A* — это алгоритм поиска пути с временной сложностью в худшем случае O(b^d). Под коэффициентом ветвления дерева b подразумевается количество соседей/потомков каждого узла. Кроме того, при расчете временной сложности учитывается глубина оптимального решения d. По сравнению с методами «слепого» поиска, такими как поиск в ширину или глубину, алгоритм A* более эффективен, когда $h(n)$ дает хорошие оценки.

Пространственная сложность алгоритма A* тоже равна O(b^d), поскольку все узлы хранятся в памяти. Этим он похож на многие другие алгоритмы поиска путей и обычно выполним, если граф не очень велик. Важно

отметить, что алгоритм можно оптимизировать за счет использования таких методов, как усечение и кэширование, позволяющих уменьшить потребность в памяти.

В целом алгоритм A* вполне справляется с решением задач поиска путей и имеет хороший баланс эффективности и точности. Благодаря временной и пространственной сложностям он прекрасно подходит для многих приложений, но важно понимать его ограничения и потенциальные возможности оптимизации, чтобы получить от него максимальную пользу.

7.5.3. Оптимальность алгоритма A*

A* — популярный алгоритм поиска пути, используемый в различных приложениях, таких как игры и робототехника. Он известен своей оптимальностью: может находить кратчайший путь от начальной до конечной точки при соблюдении определенных условий. Эти условия обусловлены эвристической функцией, используемой алгоритмом.

Во-первых, эвристическая функция должна быть допустимой, то есть она никогда не должна переоценивать фактические затраты на достижение цели. Проще говоря, эвристическая функция всегда должна возвращать оценку стоимости, которая меньше или равна фактической стоимости. Это условие гарантирует, что алгоритм исследует все возможные пути достижения цели и не упустит ни одного оптимального решения.

Во-вторых, эвристическая функция должна быть согласованной (или монотонной), то есть расчетная стоимость пути от текущего узла до цели всегда должна быть меньше расчетной стоимости от любого соседнего узла до цели плюс стоимость достижения этого соседнего узла либо должна быть равна этой расчетной стоимости. Это свойство гарантирует, что алгоритм будет выбирать наиболее перспективный путь и не станет тратить время на поиск неоптимальных решений.

Выбор функции, удовлетворяющей этим условиям, может быть сложной задачей, но при их выполнении алгоритм A* гарантированно найдет кратчайший путь, поэтому его можно задействовать в различных приложениях. Таким образом, оптимальность и эффективность данного алгоритма делают его надежным инструментом, пригодным для использования во многих ситуациях.

7.5.4. Практическое применение алгоритма А*

Алгоритм А* широко используется для обхода графа и построения оптимального пути, связывающего несколько точек, называемых узлами. Ниже перечислены некоторые известные области его применения.

1. **Видеоигры.** Алгоритм А* используется в игровой индустрии и позволяет игровым персонажам легко перемещаться в сложной среде. Он содержит несколько различных компонентов, в том числе эвристическую функцию, оценивающую расстояние между текущим положением персонажа и целью, и функцию стоимости, вычисляющую стоимость перемещения из одной позиции в другую.

   При этом алгоритм учитывает препятствия, имеющиеся на игровом поле, помогая персонажу эффективно планировать свои движения. А* произвел революцию в игровой индустрии и позволил разработчикам игр создавать более захватывающие и интерактивные игровые миры, одновременно сложные и привлекательные для игроков любого уровня подготовки.

2. **Робототехника.** В автономной навигации роботы используют алгоритмы А* для поиска наиболее эффективного пути между двумя точками. Автономная навигация — важный аспект робототехники, обеспечивающий независимое функционирование роботов в различных средах.

   Эти алгоритмы используют эвристический метод, который проверяет текущее расстояние между положением робота и точкой назначения, чтобы определить наиболее эффективный путь. С помощью алгоритмов А* роботы могут перемещаться в сложных условиях и избегать возможных препятствий.

   Эта возможность чрезвычайно важна для роботов, предназначенных для работы в реальных условиях, например на производственных предприятиях, в медицинских учреждениях, а также используемых в исследованиях космоса.

3. **Интернет-маршрутизация.** Это сложный процесс, в ходе которого пакеты данных передаются через сеть взаимосвязанных компьютеров. Цель интернет-маршрутизации — передать пакеты до места назначения наиболее эффективным путем.

   Для анализа топологии сети и расчета оптимального пути используются алгоритмы А*. Определяя лучший маршрут, эти алгоритмы учитывают самые разные факторы, такие как заторы в сети, ограничения пропускной

способности и расстояние. Использование таких алгоритмов произвело революцию в способах передачи данных через Интернет и позволило обеспечить быструю и надежную связь между устройствами по всему миру.

7.5.5. Варианты алгоритма A*

A*, при всей своей популярности, не единственный алгоритм, с помощью которого можно находить оптимальный путь. Существует еще несколько алгоритмов, основанных на A*, и каждый из них имеет сильные стороны. Ниже описаны два из них.

1. *Алгоритм A* с итеративным углублением* (Iterative Deepening A*, IDA*) — улучшенная версия алгоритма A* с точки зрения потребления памяти. Для исследования большей части графа IDA* использует поиск в глубину, тем самым уменьшая объем памяти, необходимый для хранения узлов. Он сохраняет в памяти только узлы вдоль текущего пути, то есть с меньшей вероятностью исчерпает доступную память при работе с большими графами.
2. *Упрощенный алгоритм A* с ограничением потребления памяти* (Simplified Memory-bounded A*, SMA*) — еще один вариант, ограничивающий потребление памяти алгоритмом A*. По мере приближения к установленной границе выделенной памяти он начинает отбрасывать наименее перспективные узлы. Этот алгоритм можно применять для работы с большими графами в системах с ограниченными ресурсами. SMA* может помочь предотвратить замедление алгоритма или его сбой из-за использования слишком большого объема памяти.

Подводя итог, можно сказать, что, хотя A* и является популярным алгоритмом поиска пути, не нужно забывать о других алгоритмах, производных от него. Каждый алгоритм имеет сильные и слабые стороны, и в зависимости от ситуации один алгоритм может оказаться более подходящим, чем другой.

7.5.6. Сложность алгоритма поиска A*

Временная сложность алгоритма A* зависит от используемой функции эвристики. В худшем случае временная сложность алгоритма равна $O(b^d)$, где b — коэффициент ветвления, а d — глубина оптимального решения. Этот случай возникает, когда эвристика неинформативна и приводит к тому, что стоимость достижения цели одинакова для каждого узла и поиск протекает неуправляемо.

С точки зрения пространственной сложности алгоритм A* требует хранения всех узлов в памяти, что в худшем случае дает пространственную сложность, тоже равную $O(b^d)$. Такое потребление памяти может оказаться чрезмерным, особенно если дерево поиска велико. Поэтому были созданы некоторые варианты алгоритма A* с ограниченным потреблением памяти.

Чтобы расширить ваши знания, я предлагаю попробовать различные постановки задач и реализовать алгоритм A*. Этот практический опыт в сочетании с теоретическими знаниями поможет вам лучше понять особенности и потенциал данного алгоритма.

7.6. Практические задачи

Рассмотрим некоторые задачи, решение которых поможет вам лучше понять графовые алгоритмы.

Задача 1. DFS при поиске путей через лабиринт

Начнем с простой задачи поиска пути через лабиринт с помощью алгоритма DFS. В этой задаче дана сетка (двумерный массив) и нужно найти путь от начальной позиции до заданной точки. Разрешается двигаться вверх, вниз, влево и вправо, но не по диагонали.

Вот простая реализация решения на Python с использованием DFS:

```
def dfs(maze, start, goal):
    stack = [(start, [start])]
    visited = set()

    while stack:
        (vertex, path) = stack.pop()
        if vertex not in visited:
            if vertex == goal:
                return path
            visited.add(vertex)
            for neighbor in get_neighbors(maze, vertex):
                stack.append((neighbor, path + [neighbor]))

def get_neighbors(maze, position):
    # Возвращает соседнюю позицию в лабиринте, куда можно перейти
    # Предполагается, что лабиринт представлен списком списков на Python
```

```
    i, j = position
    neighbors= [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
    valid_neighbors = []

    for x, y in neighbors:
        if 0 <= x < len(maze) and 0 <= y < len(maze[0])
          and maze[x][y] == 0: # 0 обозначает свободную позицию, 1 — стену
            valid_neighbors.append((x, y))
    return valid_neighbors
```

Вы можете проверить это решение с помощью простого лабиринта (списка списков, где 0 означает свободную позицию, а 1 — стену).

Задача 2. Поиск кратчайшего пути в сетке с помощью BFS

Поиск кратчайшего пути в сетке (например, из верхнего левого угла в нижний правый) — типичный пример применения BFS. Это связано с тем, что данный алгоритм выполняет обход узлов по уровням и посещает все вершины на одном уровне, прежде чем перейти ниже. Попробуем решить задачу с помощью BFS. Постановка задачи аналогична предыдущей, но вместо DFS для поиска кратчайшего пути используем BFS.

Вот простое решение на Python:

```
from collections import deque

def bfs(grid, start, goal):
    queue = deque([([start], 0)]) # [(path, cost)]
    seen = {start: 0}

    while queue:
        path, cost = queue.popleft()
        current = path[-1]

        if current == goal:
            return path, cost

        for neighbor in get_neighbors(grid, current):
            if neighbor not in seen or cost + 1 < seen[neighbor]:
                seen[neighbor] = cost + 1
                queue.append((path + [neighbor], cost + 1))

    return None # Путь не найден

# get_neihbors — та же функция, что и в предыдущем примере
```

Вы можете проверить, как работает поиск в ширину, используя ту же сетку, что и в предыдущем примере.

Примечание: чтобы решить эти задачи с помощью алгоритма поиска A*, алгоритма Дейкстры или других графовых алгоритмов, нужно всего лишь изменить способ исследования соседей и, возможно, добавить приоритетную очередь.

Попробуйте решить эти задачи самостоятельно. Удачи вам!

Резюме

В этой главе мы открыли для себя разнообразную и сложную область теории графов и ее прикладную ветвь — графовые алгоритмы. Сначала мы познакомились с графами, описав их как математические структуры, которые состоят из узлов, также известных как вершины, и ребер, соединяющих эти узлы. Благодаря своей способности представлять множество структур и задач графы широко используются в информатике и других областях: в социальных сетях, веб-страницах, биологических и транспортных сетях и многих других.

Познакомившись с основами графов, мы перешли в область конкретных графовых алгоритмов. Мы начали с поиска в глубину (Depth-First Search, DFS) — простой, но мощной стратегии обхода или поиска узлов в структурах данных, имеющих форму дерева либо графа. Этот алгоритм использует стек (структуру данных «последним пришел — первым ушел»), чтобы вспомнить, с какой следующей вершины продолжить поиск, если окажется в тупике. Мы рассмотрели детали реализации, изучили рекурсивную и итеративную версии, а также обсудили временную и пространственную сложности алгоритма.

Затем мы перешли к поиску в ширину (Breadth-First Search, BFS), который, в отличие от DFS, использует очередь для исследования всех непосредственных соседей текущей вершины, прежде чем перейти к вершинам на следующем уровне. Этот алгоритм особенно полезен для поиска кратчайшего пути в невзвешенных графах.

После этого мы занялись алгоритмом Дейкстры — настоящим шедевром, отыскивающим кратчайший путь от заданной начальной вершины ко всем остальным вершинам графа. Алгоритм использует приоритетную очередь для выбора следующей вершины с минимальным расстоянием и обновляет

расстояния до соседних вершин. Мы увидели, что спектр его применения весьма широк: от сетевых технологий до географии и т. д.

Далее мы рассмотрели алгоритм поиска A*, использующий эвристику для оценки стоимости достижения цели из определенной вершины, что позволяет ему осуществлять поиск более целенаправленно и эффективно находить кратчайший путь во многих сценариях. Этот алгоритм отлично подходит для таких приложений, как поиск пути в играх, GPS-навигация и т. д.

Наконец, мы углубились в практические задачи, требующие применения вновь обретенных знаний. Мы увидели, как эти алгоритмы можно использовать для решения реальных задач, таких как поиск пути в лабиринте и поиск кратчайшего пути в сетке.

На протяжении всей главы мы четко осознавали важность деталей реализации: от структур данных, таких как стеки, очереди и приоритетные очереди, до стратегий маркировки посещенных вершин и методов обновления расстояний или затрат.

В заключение можно сказать, что графовые алгоритмы служат краеугольным камнем в здании информатики и предлагают эффективные решения широкого круга задач. Поняв их механику и освоив приемы реализации, вы получите мощный набор инструментов, которые, несомненно, будут играть важнейшую роль в ваших вычислительных проектах.

Помните, что путь обучения бесконечен. Продолжайте исследовать мир алгоритмов, внедрять их и оптимизировать!

Глава 8
СТРУКТУРЫ ДАННЫХ, ИСПОЛЬЗУЕМЫЕ В АЛГОРИТМАХ

Структуры данных, как вы уже наверняка знаете, являются важнейшими компонентами любого алгоритма. Они позволяют организовывать данные и управлять ими, помогая повысить эффективность алгоритмов. Другими словами, структуры данных подобны строительным блокам алгоритмов. Они формируют подходы к решению задач.

Важно отметить, что структуры данных — не просто теоретические концепции. Они имеют вполне практическое применение почти в каждой программной системе: от операционных систем до веб-приложений, от анализа данных до машинного обучения. Они влияют на производительность, сложность и эффективность наших решений. Поэтому владение структурами данных — важный навык для любого начинающего программиста или информатика.

В начале главы мы рассмотрим массивы — одну из самых простых и часто используемых структур данных. Массивы предназначены для хранения коллекций элементов одного типа, например целых чисел, чисел с плавающей запятой или символов. Они невероятно универсальны и широко применяются в различных алгоритмах, что делает их ценным инструментом, который должен иметь в своем арсенале любой программист. В первом разделе мы рассмотрим синтаксис массивов, их инициализацию и обработку. Мы также обсудим преимущества и недостатки массивов и сравним их с другими структурами данных. К концу главы вы получите полное представление о массивах и их роли в алгоритмах.

8.1. Массивы

Массив — фундаментальная структура данных, используемая для хранения коллекций элементов одного типа. Это контейнер, который может хранить последовательность элементов фиксированного размера одного типа. Массивы широко используются в программировании компьютеров

и помогают эффективно хранить и получать доступ к нескольким значениям одного типа.

Помимо хранения, массивы можно использовать для выполнения различных операций с данными: например, сортировки, поиска определенных значений и выполнения математических операций с данными.

Индексация с нуля — обычная практика во многих языках программирования, когда речь идет о массивах. Это означает, что первый элемент массива имеет индекс 0, а не 1. На первый взгляд может показаться, что это правило будет вызывать путаницу, однако на самом деле оно вполне логично, если учесть, как компьютеры хранят данные в памяти. Используя индексацию с нуля, можно легко вычислить адрес в памяти каждого элемента массива, что делает доступ к данным намного более эффективным.

Рассмотрим пример объявления и использования массива в языке Python:

```
# Объявление массива с пятью элементами
my_array = [1, 2, 3, 4, 5]

# Доступ к элементам массива
print(my_array[0]) # Выведет: 1
print(my_array[4]) # Выведет: 5

# Изменение элемента в массиве
my_array[2] = 10
print(my_array) # Выведет: [1, 2, 10, 4, 5]
```

Массивы просты и понятны, но их возможности обусловлены способностью обеспечивать постоянное время доступа ($O(1)$) к любому элементу, что делает массивы невероятно эффективными при чтении данных. Однако за эту эффективность приходится платить: массивы имеют фиксированный размер, а это означает, что добавление или удаление элементов из массива может быть дорогостоящей в вычислительном отношении операцией с временной сложностью $O(n)$ в худшем случае, где n — длина массива.

Массивы — важный компонент многих алгоритмов и служат основой для многих структур данных. Они позволяют получить эффективный доступ к данным и обрабатывать их. Даже более сложные структуры данных, такие как кучи и хеш-таблицы, полагаются на массивы как на базовую структуру для хранения данных. Понимать работу массивов и уметь обращаться с ними крайне важно.

Далее в этой главе мы исследуем особенности взаимодействия с массивами и их связь с другими структурами данных в контексте разработки алгоритмов.

Кроме того, мы рассмотрим различные типы массивов: одномерные, многомерные и динамические, — а также области их применения. Понимание компромиссов между этими массивами играет важную роль в разработке эффективных алгоритмов.

Помимо разработки и реализации алгоритмов, массивы имеют множество разных применений. Например, они часто используются в научном моделировании, анализе данных и приложениях машинного обучения. Вдобавок они широко применяются в компьютерной графике для представления изображений и других визуальных данных.

Массивы служат базовой составляющей для многих приложений в информатике. Имея полное представление о них, можно разрабатывать эффективные алгоритмы и приложения в различных областях и отраслях.

8.1.1. Свойства массивов и основы их применения

Чтобы сделать наше обсуждение более глубоким, обсудим еще несколько свойств массивов и наиболее частые варианты их использования.

1. **Многомерные массивы.** Массивы могут иметь несколько измерений, что позволяет хранить более сложные данные. В отличие от одномерных массивов, полезных для хранения простых данных, многомерные дают возможность хранить более сложные структуры, такие как матрицы, таблицы и сетки. Типичный пример — двумерный массив, в котором каждый элемент представляет другой одномерный массив, часто называемый матрицей.

   Так, в двумерном массиве размером 3 × 3 каждый из трех элементов основного массива сам является массивом, состоящим из трех элементов. Многомерные массивы — важная концепция во многих приложениях, таких как обработка изображений, научные вычисления и многое другое. Они предоставляют эффективный способ хранения и управления большими объемами данных и особенно полезны в приложениях, где данные естественным образом организованы в табличную структуру.

   #### Пример

   ```
   # Объявление двумерного массива с тремя элементами,
   # каждый из которых сам является массивом с тремя элементами
   my_2d_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   ```

```
# Доступ к элементам двумерного массива
print(my_2d_array[0][0]) # Выведет: 1
print(my_2d_array[2][2]) # Выведет: 9

# Изменение элемента двумерного массива
my_2d_array[1][1] = 10
print(my_2d_array) # Выведет: [[1, 2, 3], [4, 10, 6], [7, 8, 9]]
```

2. **Сортировка массива.** Это распространенная операция, используемая во многих алгоритмах информатики. Она крайне важна, поэтому нужно понимать ее основы, чтобы иметь возможность создавать эффективные алгоритмы.

   Python — один из языков программирования, предоставляющих встроенные методы сортировки массива. Их можно использовать для сортировки массивов в порядке возрастания или убывания, в зависимости от потребностей алгоритма. Помимо встроенных методов, существует множество сторонних библиотек, реализующих более продвинутые алгоритмы сортировки, такие как быстрая сортировка и сортировка слиянием.

   Для конкретных типов данных важно уметь выбирать правильный алгоритм сортировки, поскольку одни алгоритмы более эффективны, чем другие. В целом понимание сортировки массивов является фундаментальным навыком для любого информатика, а встроенные методы Python служат отличной отправной точкой, с которой можно начинать изучение этой важной концепции.

Пример

```
# Объявление массива
my_array = [5, 3, 1, 4, 2]

# Сортировка массива в порядке возрастания
my_array.sort()
print(my_array) # Выведет: [1, 2, 3, 4, 5]

# Сортировка массива в порядке убывания
my_array.sort(reverse=True)
print(my_array) # Выведет: [5, 4, 3, 2, 1]
```

3. **Поиск в массиве.** При работе с массивами часто необходимо найти определенное значение. В Python имеется встроенная операция для этой задачи, реализованная как ключевое слово `in`. Оно выполняет линейный поиск в массиве, то есть последовательно проверяет каждый элемент массива, пока не найдет совпадения.

Однако линейный поиск в очень больших массивах может быть неэффективным. В таких случаях могут потребоваться более совершенные алгоритмы поиска. Однако использование ключевого слова `in` — это простой и эффективный способ поиска значений в массиве, подходящий во многих случаях. Кроме того, стоит отметить, что Python поддерживает множество других операций с массивами, таких как сортировка, фильтрация и отображение.

Эти операции могут пригодиться в самых разных приложениях — от анализа данных до машинного обучения. Итак, если вы работаете с массивами в Python, то обязательно выделите время на изучение всего спектра возможностей, которые может предложить этот язык.

Пример

```
# Объявление массива с пятью элементами
my_array = [1, 2, 3, 4, 5]

# Поиск значения в массиве
if 3 in my_array:
    print("Значение найдено!") # Выведет: Значение найдено!
if 6 in my_array:
    print("Значение найдено!")
```

4. **Получение срезов массива.** Эта функция Python позволяет извлечь часть массива и создать из нее новый.

   Функция может пригодиться при работе с большими наборами данных, позволяя сосредоточиться на той части данных, которая имеет отношение к анализу. С помощью этой функции можно извлекать как диапазон значений, так и одно значение.

   Эту функцию также можно использовать для создания новых массивов, которые являются подмножествами исходного массива, что может пригодиться при работе со сложными структурами данных. В целом извлечение среза массива — мощный инструмент, помогающий более эффективно работать с массивами в Python.

Пример

```
# Объявление массива
my_array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Извлечение среза с индексами от 2 до 5
sub_array = my_array[2:6]
print(sub_array) # Выведет: [2, 3, 4, 5]
```

```
# Извлечение среза от начала до индекса 3
start_to_index = my_array[:4]
print(start_to_index) # Выведет: [0, 1, 2, 3]

# Извлечение среза от индекса 4 до конца массива
index_to_end = my_array[4:]
print(index_to_end) # Выведет: [4, 5, 6, 7, 8, 9]
```

Функция извлечения среза массива очень удобна во многих случаях, когда нужно работать с частью массива. Кроме того, она демонстрирует возможности Python по предоставлению лаконичных и интуитивно понятных инструментов для решения типичных задач.

Массивы просты и часто воспринимаются как нечто само собой разумеющееся. Тем не менее они лежат в основе многих фундаментальных алгоритмов и служат первым средством, позволяющим получить представление о более сложных структурах данных и способах их применения. Не забывайте всегда учитывать возможность использования массивов в ваших алгоритмах, поскольку они предоставляют гибкие и эффективные способы хранения данных и доступа к ним.

Итак, мы довольно близко познакомились с массивами, сделав нашу первую остановку на пути к пониманию различных структур данных, используемых в алгоритмах. Надеемся, что этот материал помог вам получить достаточно полное представление о массивах. Если у вас есть какие-либо вопросы, которые вы хотели бы обсудить, то не стесняйтесь спрашивать. Напишите нам на https://www.cuantum.tech/contact/. Приятного обучения!

8.2. Связанные списки

Связанные списки — очень интересная и мощная структура данных, которую можно использовать для решения множества задач. Они организуют элементы последовательно, причем каждый из них имеет указатель на следующий в списке. Этот простой, но элегантный подход позволяет эффективно вставлять и удалять элементы, что делает их идеальными для использования в ситуациях, требующих применения динамических структур данных.

Помимо того что они эффективны, связанные списки еще и невероятно гибкие. Они могут быть односвязанными, когда каждый элемент указывает на следующий, или двусвязанными, когда каждый элемент указывает не только на следующий, но и на предыдущий элемент в списке. Благодаря

такой гибкости спектр их применения очень широк: от реализации стеков и очередей до создания сложных структур данных, таких как деревья и графы.

В целом связанные списки являются фундаментальной концепцией информатики и должны быть в арсенале каждого программиста. И новички, и опытные разработчики должны понимать принципы работы связанных списков и способы их использования. Это может помочь в решении задач. Поэтому в следующий раз, когда вы встанете перед вопросом выбора структуры данных, не забудьте рассмотреть связанные списки как вариант решения!

В связанном списке каждый элемент содержится в узле, состоящем из двух элементов, таких как:

- **данные** — содержат значение, хранящееся в узле;
- **указатель на следующий элемент** — содержит ссылку (или указатель) на следующий узел в цепочке.

Первый узел связанного списка называется *головой*. Последний узел связанного списка, или *хвост*, содержит пустой указатель, который служит признаком конца списка.

Вот простая схема, иллюстрирующая эту концепцию:

```
[Голова   | ] --> [Данные | ] --> [Данные | ] --> [Данные |  Пустой указатель ]
```

Простой класс `Node` и класс `LinkedList` в Python можно определить так:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = Node()
```

Класс `LinkedList` хранит ссылку на головной узел. Изначально этот узел не содержит никаких данных.

Теперь допишем функции для добавления элементов в конец (`append`) и вывода содержимого списка:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.head = Node()

    def append(self, data):
        new_node = Node(data)
        cur = self.head
        while cur.next:
            cur = cur.next
        cur.next = new_node

    def display(self):
        elems = []
        cur_node = self.head
        while cur_node.next:
            cur_node = cur_node.next
            elems.appenf(cur_node.data)
        print(elems)
```

Вот как можно создать связанный список и добавить в него элементы:

```
my_list = LinkedList()

my_list.append(1)
my_list.append(2)
my_list.append(3)

my_list.display() # Выведет: [1, 2, 3]
```

Это суть связанного списка! В отличие от массивов, связанные списки являются динамическими структурами данных и более эффективны в некоторых операциях, таких как вставка и удаление. Однако они используют больше памяти для хранения данных из-за необходимости хранить дополнительные указатели и имеют более низкую скорость доступа к отдельным элементам.

8.2.1. Другие типы связанных списков

Двусвязные списки

В информатике связанный список — это структура данных, состоящая из последовательности узлов. Каждый узел содержит элемент данных и ссылку на следующий узел. Однако обход связанного списка от конца к началу невозможен без дополнительной информации.

В подобных случаях предпочтительнее использовать двусвязные списки! Каждый узел в таком списке имеет ссылку не только на следующий, но и на предыдущий узел. Это позволяет проводить обход в обоих направлениях, что может пригодиться в некоторых приложениях. Например, такая организация списка может упростить манипулирование последовательностями данных как в прямом, так и в обратном направлении.

Однако двусвязный список требует больше памяти, чем односвязный, из-за необходимости хранить дополнительные ссылки. Но, несмотря на этот недостаток, двусвязные списки широко используются в различных приложениях и являются важной концепцией, которую должны изучать начинающие программисты.

Пример

Каждый узел двусвязного списка содержит элемент и две ссылки, указывающие на следующий и предыдущий узлы.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None

class DoubleLinkedList:
    def __init__(self):
        self.head = Node()
```

Чтобы добавить элемент в список, понадобится метод, соответствующим образом обрабатывающий ссылки `next` и `prev`.

Циклические связанные списки

Циклический связанный список — это разновидность связанного списка, в котором ссылка в последнем узле содержит указатель на головной узел, а не `None`. В результате он образует круговую цепочку, полезную в различных приложениях. Например, планировщик задач операционной системы обычно хранит данные в таких циклических списках и осуществляет управление задачами «по кругу».

Кроме того, с помощью циклических связанных списков в компьютерной графике можно создавать фигуры с замкнутой границей, такие как дуги

или круги. Вдобавок их можно использовать в различных структурах данных, таких как стеки и очереди, для более эффективной реализации алгоритмов.

Несмотря на свою полезность, циклические связанные списки сложнее обычных связанных списков, поэтому за ними нужно тщательно следить, чтобы исключить вероятность бесконечного зацикливания и избежать утечек памяти. Однако при правильной реализации циклические связанные списки могут стать мощным инструментом в арсенале программиста.

Пример

В циклическом связанном списке ссылка `next` в последнем узле указывает на первый узел.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = Node()

    def append(self, data):
        if not self.head:
            self.head = Node(data)
            self.head.next = self.head
        else:
            new_node = Node(data)
            cur = self.head
            while cur.next != self.head:
                cur = cur.next
            cur.next = new_node
            new_node.next = self.head
```

Эта реализация создает новый узел. Если список пустой (`self.head` имеет значение `None`), то в `self.head` записывается ссылка на новый узел, а в ссылку `next` нового узла — указатель на сам этот узел. Если список не пустой, то выполняется переход к концу списка (когда `cur.next` ссылается на `self.head`) и указатели `next` последнего и нового узлов корректируются так, чтобы сохранить циклическую структуру.

8.3. Стеки и очереди

Стеки и очереди — два типа структур данных, широко используемых в информатике. Они во многом похожи на массивы и связанные списки, но имеют некоторые важные различия, которые делают их особенно полезными в определенных ситуациях.

Стек — структура данных, которая хранит коллекцию элементов и поддерживает две основные операции: `push` (добавляет элемент на вершину стека) и `pop` (удаляет верхний элемент с вершины стека). Более подробно стеки рассмотрим в подразделе 8.3.1.

Очередь — структура данных, которая хранит коллекцию элементов и поддерживает две основные операции: `enqueue` (добавляет элемент в конец очереди) и `dequeue` (удаляет первый элемент из очереди). Более подробно очереди рассмотрим в подразделах 8.3.2 и 8.3.3.

Поняв, чем стеки и очереди отличаются от массивов и очередей, вы сможете выбрать ту структуру данных, которая лучше всего соответствует вашим потребностям, и написать более эффективный и элегантный код.

8.3.1. Стеки

Стеки следуют принципу «последним пришел — первым ушел» (Last-In-First-Out, LIFO). Это означает, что последний элемент, добавленный в стек, будет удален первым. Стеки похожи на стопку книг. Вы кладете книгу наверх стопки (операция `push`), и единственная книга, которую можно взять из стопки, — та, что находится вверху (операция `pop`).

Стеки — важная структура данных, используемая в информатике. В языках программирования, операционных системах и других программных приложениях стеки часто служат для управления данными. Например, функция отмены в текстовом редакторе может использовать стек для хранения предыдущих состояний документа и позволять отменять изменения шаг за шагом в обратном порядке.

Помимо операций `push` и `pop`, стеки поддерживают другие операции, такие как `peek`, которая позволяет просмотреть элемент на вершине стека, не удаляя его, и `isEmpty`, сообщающую, пуст ли стек.

В целом стеки — простая, но мощная структура данных, широко используемая в информатике. Они понятны, легко реализуются и имеют множество вариантов практического применения, что делает их важным инструментом в арсенале любого программиста или разработчика программного обеспечения.

Вот простая реализация стека на Python:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if len(self.stack) < 1:
            return None
        return self.stack.pop()

    def size(self):
        return len(self.stack)
```

Операция push добавляет элемент в конец списка, а операция pop удаляет последний элемент из списка.

8.3.2. Очереди

В отличие от стеков, очереди следуют принципу «первым пришел — первым ушел» (First-In-First-Out, FIFO).

Очереди широко распространены в реальном мире. Например, представим цепочку посетителей в кафетерии. Первый человек в очереди будет обслужен первым, а новые посетители будут вставать в конец очереди. Список ожидания в кабинете врача тоже можно рассматривать как очередь, в которой первый человек, пришедший на прием, будет первым, кого встретит врач. Это соответствует операциям enqueue и dequeue, выполняемым с очередями.

Очереди можно использовать и в компьютерных приложениях. Например, когда компьютер печатает несколько документов одновременно, он использует очередь, чтобы гарантировать печать документов в правильном порядке. Аналогично, когда веб-сервер получает сразу несколько запросов, он ставит их в очередь и обрабатывает в том порядке, в каком они были получены.

Вот простая реализация очереди на Python:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    dev dequeue(self, item):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    def size(self):
        return len(self.queue)
```

В этой реализации очереди операция `enqueue` добавляет элемент `item` в конец списка, а `dequeue` удаляет первый элемент из списка.

Стеки и очереди играют важную роль во многих алгоритмах, особенно в тех, которые связаны с обходом графа. Например, поиск в глубину использует стек, а поиск в ширину — очередь.

Кроме того, стоит отметить, что Python предоставляет встроенные реализации стеков и очередей, имеющие довольно широкие возможности. В простых случаях в роли стеков можно использовать обычные списки с методами `append` и `pop`, как показано выше. Очереди в Python реализованы в модуле `queue`, который содержит различные классы, такие как `Queue`, `LifoQueue` и `PriorityQueue`, для разных вариантов использования.

Мы рассмотрели основы стеков и очередей. Они широко используются в различных задачах и алгоритмах, и понимание того, как они работают, определенно поможет вам при решении алгоритмических задач. Продолжая исследовать мир алгоритмов и структур данных, вы будете сталкиваться с ними довольно часто.

8.3.3. Приоритетные и двунаправленные очереди

Теперь углубимся в исследование приоритетных и двунаправленных очередей, дополняющих многообразие структур данных, подпадающих под категорию стеков и очередей.

Приоритетные очереди

Приоритетные очереди — очень полезная структура данных, которую можно использовать в различных приложениях в области информатики. Это особый тип очереди, в которой каждому элементу присваивается приоритет — числовое значение, определяющее важность элемента. Например, если вы создаете программу, которая должна планировать решение задач, то можете назначить более высокий приоритет более срочным или важным задачам.

Такие очереди можно реализовать несколькими способами, например с помощью массива или связанного списка. Один из наиболее распространенных способов реализации приоритетной очереди — использование кучи, то есть двоичного дерева, удовлетворяющего свойству кучи. Согласно этому свойству значение любого узла в дереве больше значений его дочерних узлов или равно им. Это свойство гарантирует, что элемент с наивысшим приоритетом всегда находится на вершине кучи.

Элементы с одинаковым приоритетом в приоритетной очереди обслуживаются в соответствии с их порядком в ней. Это означает, что если два элемента имеют одинаковый приоритет, то первым будет обслужен тот, который был добавлен в очередь первым. Это позволяет объективно и эффективно обрабатывать элементы с одинаковым приоритетом.

В целом приоритетные очереди — мощный инструмент, который можно использовать в широком спектре приложений. Понимая, как реализуются и работают приоритетные очереди, вы сможете повысить эффективность своих программ.

Встроенная реализация приоритетной очереди в Python доступна в модуле `heapq`. Вот простой пример:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, priority, item):
        heapq.heappush(self.queue, (-priority, item)) # heapq реализует только
                                                      # неубывающую кучу

    def dequeue(self):
        if len(self.queue) < 1:
            return None
    return heapq.heappop(self.queue)[1] # Вернуть только сам элемент,
                                        # без приоритета
```

Двунаправленные очереди

Двунаправленные очереди (double-ended queue, deque) — структура данных, позволяющая вставлять и удалять элементы с любого конца. В отличие от простой очереди, которая следует правилу FIFO («первым пришел — первым ушел»), двунаправленные очереди позволяют выполнять более гибкую обработку данных.

В Python имеется встроенная реализация двунаправленной очереди в виде эффективного и удобного объекта `deque` в модуле `collections`. Этот объект помогает проще и эффективнее обрабатывать данные, не нанося ущерба производительности. Двунаправленные очереди можно использовать и для реализации стека или простой очереди, что делает их универсальным инструментом для обработки данных при программировании на Python.

Рассмотрим пример:

```
from collections import deque

d = deque()

# Метод append() добавляет элементы в очередь с правого конца
d.append('B')
d.append('C')

# Метод appendleft() добавляет элементы в очередь с левого конца
d.appendleft('A')

print("Очередь: ", d) # Выведет: Очередь: deque(['A', 'B', 'C'])

# Метод pop() удаляет элементы с правого конца
d.pop()
print("Очередь после удаления элементов справа: ", d)
# Выведет: Очередь после удаления элементов справа: deque(['A', 'B'])

# Метод popleft() удаляет элементы с левого конца
d.popleft()
print("Очередь после удаления элементов слева: ", d)
# Выведет: Очередь после удаления элементов слева: deque(['B'])
```

Приведенные выше примеры реализуют некоторые базовые возможности этих структур. Но помните, что в практических приложениях чаще используются встроенные структуры данных или библиотеки Python, которые предлагают более широкие возможности и обеспечивают оптимизированную производительность.

8.4. Деревья и графы

Деревья и графы — нелинейные структуры данных, широко используемые для моделирования связей между различными данными. Деревья, например, очень полезны для представления иерархических данных, таких как генеалогические деревья, компьютерные файловые системы и организационные диаграммы.

Графы, в свою очередь, можно использовать для моделирования сложных отношений между различными объектами, такими как социальные сети, транспортные сети и др. Деревья и графы позволяют создавать более сложные и осмысленные представления, в отличие от простых линейных структур данных, таких как массивы, стеки или очереди.

Рассмотрим эти структуры более подробно.

8.4.1. Деревья

Дерево — структура данных, состоящая из иерархически организованного набора связанных узлов. Первый (самый верхний) узел называется корневым и является начальной точкой дерева. С корнем связаны дополнительные узлы, которые могут иметь дочерние узлы. Узлы, не имеющие родительских элементов, называются корнями, а узлы, не имеющие дочерних элементов, — листьями. Каждый узел дерева имеет уникальный путь от корня до самого себя, который называется ветвью.

Особый тип дерева — *двоичное дерево* — ограничивает количество дочерних элементов узла. В частности, в двоичном дереве любой узел может иметь не более двух дочерних элементов. Эти деревья широко используются в информатике, поскольку ими легко манипулировать. Ограничивая количество дочерних элементов, двоичные деревья обеспечивают более высокую скорость поиска, чем деревья других типов, поэтому идеально подходят для хранения и извлечения данных в компьютерных приложениях.

Создадим простое двоичное дерево:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

root = Node(1)
```

```
root.left = Node(2)
root.right = Node(3)

root.left.left = Node(4)
root.left.right = Node(5)
```

Код в этом примере создает двоичное дерево с пятью узлами: 1 — корневой узел, 2 и 3 — дочерние элементы корня, а 4 и 5 — дочерние элементы узла, содержащего 2.

Двоичные деревья поиска

Двоичное дерево поиска (Binary Search Tree, BST) — структура данных, используемая для представления данных с иерархической организацией, в которой все узлы соответствуют следующему свойству: левый дочерний узел меньше родительского, а правый дочерний — больше. Это свойство позволяет выполнять поиск, вставку и удаление за время $O(\log n)$, что делает двоичные деревья весьма эффективными.

Двоичные деревья обладают и другими преимуществами. Например, они просты в реализации, и их можно использовать в различных приложениях, выполняя, помимо прочего, сортировку, поиск и сжатие данных. Более того, двоичные деревья можно применять в различных языках программирования, что делает их универсальной и широко используемой структурой данных. В целом двоичные деревья — отличный инструмент управления данными, на который стоит обратить внимание при разработке приложений, где важны высокая скорость поиска, вставка и удаление.

Деревья AVL

Дерево AVL — это самобалансирующееся двоичное дерево поиска, названное по именам его изобретателей Адельсона-Вельского (Adelson-Velsky) и Лэндиса (Landis). Дерево AVL стремится поддерживать высоту двух дочерних поддеревьев любого узла так, чтобы они отличались не более чем на единицу, гарантируя сбалансированность дерева.

Поддержание дерева в сбалансированном состоянии гарантирует, что временная сложность операций поиска, вставки и удаления в худшем случае составит $O(\log n)$, где n — количество узлов в дереве. Это означает, что время, необходимое для выполнения этих операций, растет логарифмически по мере увеличения количества узлов, что делает деревья AVL быстрой и эффективной структурой для хранения и извлечения данных.

Красно-черные деревья

Красно-черное дерево — разновидность самобалансирующегося двоичного дерева поиска, сбалансированность которого поддерживается за счет использования узлов с цветовой кодировкой. Каждый узел дерева содержит дополнительный бит, известный как бит цвета, который может обозначать красный или черный цвет. Этот бит используется для балансировки дерева во время вставок и удалений, благодаря чему улучшается временная сложность.

Дерево поддерживает себя в сбалансированном состоянии, гарантируя таким образом высокую эффективность операции поиска. Благодаря этому оно становится структурой, оптимально подходящей для хранения и извлечения больших объемов данных. Кроме того, использование узлов с цветовой кодировкой усложняет структуру дерева, позволяя применять его в широком спектре приложений, таких как сетевая маршрутизация, индексирование баз данных и т. д.

В целом красно-черные деревья — универсальная структура данных, которая позволяет найти баланс между эффективностью и сложностью, что делает ее важным инструментом в арсенале любого разработчика ПО или информатика.

Пример

Ниже перечислены основные правила построения красно-черных деревьев:

- каждый узел либо красный, либо черный;
- корневой узел — черный;
- все листья (конечные узлы) — черные;
- если узел красный, то оба его потомка — черные;
- все пути от узла к его листьям-потомкам содержат одинаковое количество черных узлов.

```
# Пример красно-черного дерева:
# каждый узел представлен списком [color, value], где
# 1 в поле color означает черный цвет, а 0 — красный

tree = [
    [1, 11],
    [[1, 2], [0, 7], [1, 14]]
]
```

Здесь корень дерева — 11 (черный) — имеет трех потомков: 2 (черный), 7 (красный) и 14 (черный).

В-деревья

В-деревья — разновидность самобалансирующихся деревьев поиска, которые часто используются в базах данных и файловых системах. Эти деревья позволяют эффективно выполнять операции вставки, удаления и поиска, поэтому идеально подходят для крупномасштабных приложений. В В-дереве порядка m каждый узел может иметь не более $m - 1$ ключей и m дочерних узлов.

Благодаря такой структуре дерево остается сбалансированным, и это позволяет поддерживать высокую производительность с течением времени. Кроме того, В-деревья можно использовать для обработки больших объемов данных, что делает их ценным ресурсом для приложений, управляющих большими базами данных или файловыми системами. В целом В-деревья — полезный инструмент как для разработчиков, так и для специалистов по обработке данных и благодаря своей универсальности и эффективности часто используются в самых разных приложениях.

Пример

В В-дереве все листья находятся на одной высоте (от корня их отделяет одинаковое количество ребер).

```
# Пример B-дерева:
# Каждый узел - это список значений, внутренние списки - это поддеревья
tree = [
    [10, 20]
    [[5, 7], [12, 15, 18], [25, 30]]
]
```

Здесь корень дерева — [10, 20], он разбивает дерево на три поддерева со значениями меньше 10, между 10 и 20 и больше 20 соответственно.

8.4.2. Графы

Граф — фундаментальное понятие информатики. В главе 7 мы обсуждали, что граф содержит конечный набор вершин (узлов) и соединяющих их ребер. Ребра могут быть ориентированными или неориентированными и обычно используются для представления связей между различными частями данных. В отличие от деревьев, представляющих похожую структуру данных, графы могут иметь циклы. Это означает, что, следуя по ребрам, можно перемещаться от одного узла к другому и в итоге вернуться к начальному узлу.

В информатике существует множество способов представления графов, но один из наиболее распространенных — использование списков смежности. Этот подход предполагает создание списка для каждой вершины графа, содержащего все остальные вершины, связанные с ней ребрами. Используя данный метод, можно быстро определить, какие узлы прилегают к определенной вершине, что может пригодиться при выполнении определенных типов вычислений или при анализе графа в целом.

Вот как можно представить простой граф, используя список смежности в Python:

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_vertex(self, node):
        self.adjacency_list[node] = []

    def add_edge(self, node1, node2):
        self.adjacency_list[node1].append(node2)
        self.adjacency_list[node2].append(node1)

graph = Graph()

graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")

graph.add_edge("A", "B")
graph.add_edge("A", "C")
```

Здесь мы создали граф с тремя узлами (A, B и C) и двумя ребрами (соединяющими A и B, а также A и C).

Эти простые примеры демонстрируют структуру деревьев и графов и их возможную реализацию на Python. Однако в реальных приложениях вы можете использовать более сложные структуры с дополнительными функциями и методами.

Помните: выбор структуры всегда должен основываться на характере стоящей перед вами задачи. Некоторые задачи вполне можно решить с помощью простого массива или связанного списка. Но есть задачи, для решения которых могут понадобиться более сложные структуры, такие как деревья

и графы. Понимание свойств каждой структуры и потенциальных выгод, которые она может дать, — важная часть разработки эффективных алгоритмов.

Графы: взвешенные и невзвешенные, циклические и ациклические

В подразделе 8.4.2 мы видели простой пример графа, но, как вы уже знаете из главы 7, графы могут быть гораздо более сложными и разнообразными. Например, они могут быть взвешенными, когда каждое ребро имеет определенную стоимость (вес), или невзвешенными, когда все ребра имеют одинаковую стоимость (вес). Эта характеристика позволяет добавить больше измерений в граф и получить более полное представление о данных.

Графы могут иметь разную структуру. Они могут быть циклическими — в таком графе можно, перемещаясь по ребрам от одной вершины к другой, вернуться к началу. Они могут быть ациклическими — в таком графе нельзя вернуться к началу. Это различие может существенно повлиять на тип анализа, который можно провести с помощью графа.

Кроме того, графы могут быть ориентированными или неориентированными. В ориентированных ребра имеют определенное направление, тогда как в неориентированных — нет. Это различие тоже может повлиять на тип проводимого анализа.

В целом графы могут быть невероятно разнообразными и сложными и иметь различные характеристики и структуру, благодаря чему их можно использовать в разных видах анализа.

Ориентированные и неориентированные графы

Графы — базовый инструмент информатики и математики. Для полноты картины напомним, в чем разница между ориентированными и неориентированными графами.

Ребра в ориентированных графах, также называемых орграфами, имеют определенное направление. Это означает, что вдоль ребра можно переместиться от одной конкретной вершины к другой, но не наоборот.

Ребра в неориентированных графах не имеют определенного направления и фактически являются двунаправленными. Это означает, что вдоль ребер можно перемещаться между двумя вершинами в обе стороны и не имеет значения, какая вершина является начальной, а какая — конечной.

Понимание различий между этими двумя типами графов важно во многих областях применения, например в сетевом анализе или алгоритмах поиска пути.

Пример

В Python граф можно представить с помощью словаря, каждый ключ которого — это узел графа, а соответствующее значение — список узлов, к которым можно получить доступ из этого узла.

```
# Пример ориентированного графа
directed_graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Пример неориентированного графа
undirected_graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

Если в ориентированном графе существует путь от `A` к `B`, это не означает, что существует путь от `B` к `A`. Но в неориентированном графе все не так: если существует путь от `A` к `B`, то должен быть путь и от `B` к `A`.

Связные и несвязные графы

В связном графе все вершины соединены друг с другом, то есть между каждой парой вершин есть путь, независимо от того, насколько они удалены друг от друга. В несвязном графе есть вершины, которые не связаны с другими, что делает их недоступными из определенных вершин.

Таким образом, определенные части графа могут быть недоступны или недостижимы из некоторых других частей, что может вызывать трудности при анализе графа или его использовании в различных целях.

Планарные и непланарные графы

Планарный (плоский) граф можно нарисовать на плоскости, не используя пересечение ребер, то есть его можно представить в двумерном пространстве без пересекающихся линий. В непланарных графах, напротив, есть хотя бы одна пара пересекающихся ребер, из-за чего их невозможно изобразить на плоскости, не используя пересечение линий.

Таким образом, планарность графа является важным свойством, определяющим особенности визуального представления, и часто используется в различных областях, таких как информатика, математика и инженерия, для анализа и решения задач, связанных с сетями и связностью.

Подытожим. Существуют разные типы деревьев и графов. Каждый из них имеет уникальные достоинства, недостатки и используется по-своему. С одной стороны, двоичные деревья прекрасно подходят для операций поиска и сортировки, а сбалансированные — могут снизить временную сложность этих операций. С другой стороны, графы могут помочь представить сложные отношения между объектами и использоваться для решения таких задач, как оптимизация маршрутов и анализ социальных сетей.

Приступая к разработке алгоритма, важно выбрать правильную структуру данных, чтобы эффективно решать сложные вычислительные задачи. Понимая сильные и слабые стороны различных видов деревьев и графов и исходя из конкретных ограничений и требований решаемой задачи, вы сможете принимать более обоснованные решения о выборе структуры данных. Это позволит вам оптимизировать алгоритмы и повысить производительность ваших программ.

8.5. Практические задачи

Предложенные ниже задачи позволят вам применить на практике ваши новые знания о структурах данных, используемых в алгоритмах.

Задача 1. Массивы: подмассив с максимальной суммой

Для заданного целочисленного массива `nums` найдите непрерывный подмассив (содержащий хотя бы одно число), имеющий наибольшую сумму, и верните его сумму.

Для решения этой задачи можно воспользоваться алгоритмом Кадане. Вот решение на Python:

```
def max_subarray(nums):
    if not nums:
        return 0

    curr_sum = max_sum = nums[0]

    for num in nums[1:]:
        curr_sum = max(num, curr_sum + num)
        max_sum = max(max_sum, curr_sum)

    return max_sum

print(max_subarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])) # Выведет: 6
```

Задача 2. Связанные списки: перестановка элементов списка в обратном порядке

Для заданной головы односвязного списка переставьте элементы списка в обратном порядке и верните получившийся список.

Эту проблему легко решить, итеративно меняя местами указатели в связанном списке.

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def reverseList(head):
    prev = None
    curr = head
    while curr:
        next_temp = curr.next
        curr.next = prev
        curr = next_temp
    return prev
```

Задача 3. Стеки: проверка парности скобок

Для заданной строки `s`, содержащей только символы `'('`, `')'`, `'{'`, `'}'`, `'['` и `']'`, определите допустимость входной строки. Входная строка считается допустимой, если каждая открывающая скобка имеет соот-

ветствующую закрывающую, причем оба типа скобок должны закрываться в правильном порядке.

Эту задачу можно решить с помощью стека.

```
def isValid(s):
    stack = []
    mapping = {")": "(", "}": "{", "]", "["}
    for char in s:
        if char in mapping:
            top_element = stack.pop() if stack else '#'
            if mappimg[char] != top_element:
                return False
        else:
            stack.append(char)
    return not stack

print(isValid("()[]{}")) # Выведет: True
```

Задача 4. Деревья: максимальная глубина двоичного дерева

Определите максимальную глубину заданного корня двоичного дерева — количество узлов в самом длинном пути от корневого узла до самого дальнего листового узла.

Эту задачу можно решить с помощью алгоритма поиска в глубину (Depth-First Search, DFS).

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

def maxDepth(root):
    if root is None:
        return 0
    else:
        left_height = maxDepth(root.left)
        right_height = maxDepth(root.right)
        return max(left_height, right_height) + 1
```

Обязательно найдите время, чтобы разобраться в этих задачах и их решениях. Только практикуясь, вы сможете овладеть навыком использования этих структур данных. Успехов!

Резюме

Наше исследование структур данных в этой главе началось с массивов — самой простой и часто используемой структуры. Массивы помогают понять работу более сложных структур и многих алгоритмов. Они позволяют эффективно хранить данные и получать к ним доступ по их индексам за время O(1). Однако у массивов есть ограничения, например фиксированный размер после объявления и неэффективность таких операций, как вставка и удаление.

Затем мы перешли к связанным спискам — динамической и гибкой структуре данных, устраняющей некоторые недостатки массивов. Связанные списки прекрасно подходят для использования в сценариях, где количество данных неизвестно заранее или данные требуется часто вставлять или удалять. Мы исследовали различные типы связанных списков, в том числе односвязные, двусвязные и циклические списки, каждый из которых имеет уникальные особенности и варианты использования.

Затем мы познакомились со стеками и очередями. Эти структуры данных играют важную роль во многих областях информатики и программирования — от управления памятью до асинхронной обработки задач. Основное различие между этими двумя структурами заключается в порядке удаления элементов: стеки следуют принципу «последним пришел — первым ушел» (Last-In-First-Out, LIFO), а очереди — принципу «первым пришел — первым ушел» (First-In-First-Out, FIFO).

Наконец, мы погрузились в еще более сложные структуры — деревья и графы. Они незаменимы при работе с иерархическими и взаимосвязанными данными соответственно. Деревья с их нелинейной иерархической структурой полезны в таких случаях, как индексация базы данных и быстрая сортировка. Мы познакомились с двоичными деревьями, деревьями двоичного поиска, деревьями AVL, красно-черными и B-деревьями. Графы, отражающие отношения между парами объектов, лежат в основе социальных сетей, систем рекомендаций и многих других сложных алгоритмических задач. Графы могут быть ориентированными и неориентированными, взвешенными и невзвешенными, циклическими и ациклическими, связанными и несвязанными, планарными и непланарными в зависимости от конкретных требований.

Мы рассмотрели не только теоретические основы и детали реализации каждой структуры данных, но и различные практические задачи. Они позволили

применить теоретические знания и по-настоящему понять сильные и слабые стороны каждой структуры данных.

Подводя итог, важно отметить, что ни одна структура данных сама по себе ничем не лучше других. Каждая имеет сильные и слабые стороны и подходит для решения определенных типов задач. Опытный программист понимает эти нюансы и способен выбрать правильную структуру данных для поставленной задачи. Изучение материала этой главы помогло вам добавить в свой арсенал набор надежных структур данных, которые еще пригодятся в нашем увлекательном путешествии по миру алгоритмов.

Итак, дорогие читатели, продолжайте практиковаться, программировать и помните: выбор правильной структуры данных может превратить сложную задачу в решаемую!

Глава 9
МЕТОДЫ ПРОЕКТИРОВАНИЯ АЛГОРИТМОВ

Методы, которые мы будем обсуждать в данной главе, не просто позволяют решать широкий спектр сложных вычислительных задач, а служат составляющими мыслительного процесса, который при этом возникает. Эти методы представляют собой различные стратегии и подходы к разработке алгоритмов, и каждый из них занимает уникальное место в наборе алгоритмических инструментов опытного программиста.

По мере изучения этих методов мы обсудим их реальное применение в различных областях, таких как финансы, здравоохранение и игры. Мы увидим, как эти методы развивались с течением времени и как продолжают формировать современные подходы к сложным вычислительным задачам.

Благодаря материалу этой главы вы расширите свой кругозор и получите навыки, необходимые любому опытному программисту.

9.1. Рекурсия

Рекурсия — очень интересный метод решения задач, предполагающий разбиение сложной задачи на более мелкие и управляемые части. По сути, это процесс определения чего-либо с точки зрения самого себя. Например, рассмотрим древовидную структуру, в которой каждый узел имеет дочерние узлы. Чтобы выполнить обход дерева, можно многократно и рекурсивно вызывать функцию для каждого дочернего узла, пока не будет исследована вся структура.

Благодаря этой самореферентной природе рекурсия весьма эффективно решает задачи, связанные с повторением операций, и широко используется в информатике, математике и инженерии.

Рекурсию можно применять для решения и таких задач, как сортировка, рисование фракталов и анализ выражений. Универсальность и эффективность делают ее популярным методом в таких языках программирования,

как Python, Java и C++. В целом рекурсия — очень интересная концепция с широким спектром применения, и, освоив ее, можно решить многие задачи.

Рассмотрим рекурсию на примере простой функции вычисления факториала числа.

Факториал числа n (обозначается как $n!$) — это произведение всех натуральных чисел, меньших или равных n. Например, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Функцию вычисления факториала можно определить рекурсивно:

- базовый случай — $0! = 1$;
- рекурсивный случай — $n! = n \times (n - 1)!$, для $n > 0$.

Вот как эти вычисления можно реализовать на Python:

```
def factorial(n):
    # Базовый случай: факториал 0 равен 1
    if n == 0:
        return 1
    # Рекурсивный случай
    else:
        return n * factorial(n-1)

print(factorial(5)) # Выведет: 120
```

В этом примере функция `factorial` вызывает саму себя для вычисления факториала числа. Обратите внимание, как задача вычисления $n!$ разбивается на меньшую вычислительную задачу $(n - 1)!$.

В случае рекурсии мы выполняем рекурсивный вызов, чтобы решить меньший экземпляр той же задачи. Так продолжается до тех пор, пока не будет достигнут базовый случай, который можно решить напрямую, без дальнейшей рекурсии.

Помните, что всем рекурсивным функциям необходим базовый случай, предотвращающий бесконечную рекурсию, что очень похоже на наличие условия выхода в цикле. Базовый случай обычно обрабатывает простейший возможный экземпляр задачи, которую можно решить напрямую.

Для более полного понимания обсудим еще несколько важных аспектов рекурсии.

- **Хвостовая рекурсия.** Это особая форма рекурсии, когда рекурсивный вызов является последней операцией в рекурсивной функции. Другими

словами, когда возвращаемое значение рекурсивного вызова является возвращаемым значением всей функции. Одно из преимуществ хвостовой рекурсии — возможность сделать программу более эффективной благодаря способности компилятора или интерпретатора оптимизировать ее и избежать использования дополнительного пространства на стеке.

Это объясняется тем, что хвостовая рекурсия позволяет программе повторно использовать один и тот же кадр стека для каждого рекурсивного вызова, а не создавать новый кадр для каждого вызова, и тем самым избавиться от ненужных накладных расходов.

Еще одно преимущество хвостовой рекурсии — возможность облегчить чтение и понимание кода, так как при этом устраняется необходимость в явном базовом случае, что делает рекурсивную структуру функции более явной.

При этом важно отметить, что не все рекурсивные функции можно записать в форме хвостовой рекурсии, а в некоторых алгоритмах для достижения желаемого результата может потребоваться более сложная рекурсивная структура.

Вот пример функции `factorial` с хвостовой рекурсией:

```
def factorial(n, acc=1):
    # Базовый случай: факториал 0 равен 1
    if n == 0:
        return acc
    # Рекурсивный случай
    else:
        return factorial(n-1, n * acc)

print(factorial(5)) # Выведет: 120
```

В этой версии `acc` (сокращенно от accumulator) хранит текущий результат вычислений. Эта реализация более эффективна, чем предыдущая, поскольку интерпретатору не нужно хранить промежуточные результаты в памяти.

- **Косвенная рекурсия.** Косвенная рекурсия немного сложнее прямой и имеет место, когда одна функция вызывает другую, а та, в свою очередь, вызывает первую. В результате создается цикл взаимных вызовов функций. Косвенную рекурсию можно использовать по-разному, в том числе для решения задач, требующих совместной работы нескольких функций.

С ее помощью также можно создавать сложные алгоритмы, требующие множества взаимозависимых функций. В целом косвенная рекурсия может быть весьма эффективной, но требует тщательного планирования и выполнения. Это позволяет гарантировать, что все функции будут работать правильно, не попадая в бесконечный цикл и не вызывая другие ошибки.

- **Рекурсивные структуры данных.** Структура данных — это вариант организации и хранения данных в компьютере, позволяющий эффективно извлекать их и изменять. Структура данных называется рекурсивной, если она определена в терминах меньшего экземпляра структуры данных того же типа.

  Эта идея часто используется в информатике для упрощения представления сложных данных. Наиболее распространенным примером рекурсивной структуры данных может служить связанный список. Это набор узлов, каждый из которых содержит некие данные и ссылку на следующий узел. Определив связанный список в виде меньшего связанного списка, можно создать простую и эффективную структуру данных.

  Идею рекурсии можно также применить ко многим другим структурам данных, таким как двоичные деревья и графы, что позволяет представлять сложные данные простым и элегантным способом.

Несмотря на то что рекурсия предоставляет элегантный, а иногда и более понятный подход к решению задач, важно помнить и о ее недостатках.

- **Переполнение стека.** Каждый рекурсивный вызов приводит к созданию нового кадра на стеке вызовов. Если рекурсия слишком глубока (то есть когда выполняется слишком много вложенных вызовов), может возникнуть ситуация, когда в стеке вызовов не окажется места для размещения новых кадров стека, генерируемых каждым вызовом. Эта проблема особенно актуальна для языков с фиксированным максимальным размером стека, таких как C++, и известна как ошибка переполнения стека.

  Чтобы ее избежать, в таких случаях важно оптимизировать код и постараться уменьшить количество рекурсивных вызовов.

- **Избыточные вычисления.** В некоторых рекурсивных реализациях одна и та же подзадача может решаться несколько раз, что снижает эффективность. В ряде случаев подобную неэффективность можно устранить с помощью динамического программирования, предполагающего сохранение результатов промежуточных подзадач в таблице, чтобы их можно было

найти и использовать позже. Этот метод особенно полезен, когда для решения основной задачи требуется решить перекрывающиеся подзадачи, как в случае с числами Фибоначчи.

Динамическое программирование позволяет избежать повторения вычислений и значительно повысить скорость работы алгоритма. Фактически версия вычисления чисел Фибоначчи, реализованная с использованием динамического программирования, имеет временную сложность $O(n)$, тогда как простая рекурсивная реализация — $O(2\text{^}n)$, из-за чего работает намного медленнее в случае больших значений n.

Поэтому важно изыскать возможность использования динамического программирования при решении задач, связанных с рекурсивными вычислениями, особенно если одни и те же подзадачи могут встречаться несколько раз.

Рассмотрим пример:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
```

В этой версии вычисление `fibonacci(n-2)` повторяется снова и снова. При больших значениях `n` это приводит к существенному снижению эффективности.

Решить проблемы избыточных вычислений или большой глубины рекурсии, которые могут замедлить работу программы, можно с помощью мемоизации, то есть кэширования результатов предыдущих вызовов функций.

Другой вариант — использовать итеративное решение. Этот подход особенно эффективен, когда глубина рекурсии может оказаться слишком большой. Решение об использовании рекурсии вместо итераций часто зависит от специфики конкретной задачи, используемого языка и компромиссов между простотой кода и эффективностью выполнения. Тем не менее важно помнить, что владение рекурсией открывает новый способ представления задач и обогащает ваш инструментарий.

Рекурсия — фундаментальная концепция информатики, с которой вы часто будете сталкиваться в своей работе. Практикуясь все больше и больше, вы вне всяких сомнений освоите ее. Так что продолжайте исследовать ее возможности!

9.2. Итеративные подходы

В разработке алгоритмов есть два базовых подхода к решению задач: рекурсивный и итерационный. В отличие от рекурсии, основанной на вызовах функций, итерационный подход основан на циклах. Преобразование рекурсивной функции в итеративную — важнейший навык программирования, требующий хорошего понимания обоих подходов.

Конечно, рекурсия имеет преимущества, но итеративные подходы, как правило, более эффективны с точки зрения производительности и потребления памяти. Это связано с отсутствием накладных расходов на работу со стеком вызовов функций, что позволяет им обрабатывать большие объемы входных данных, не рискуя вызвать переполнения стека.

Рассмотрим итеративный подход на примере вычисления факториала. Как вы уже знаете, факториал является произведением всех положительных целых чисел, меньших или равных неотрицательному целому числу n, и обозначается как $n!$. Факториал числа можно вычислить, используя итеративный подход, перемножающий все числа от 1 до n в цикле. Этот подход не только более эффективен с точки зрения потребления памяти, но и помогает лучше понять, как применять итерации при разработке алгоритмов.

9.2.1. Итеративное вычисление факториала

```
def factorial_iterative(n):
    if n < 0:
        return "Недопустимый ввод! Факториал для отрицательных значений
          не определен."
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5)) # Выведет: 120
```

Функция `factorial_iterative` инициализирует `result` числом `1`. Затем запускает цикл от `1` до `n` (включительно) и в каждой итерации умножает `result` на каждое целое от `1` до `n`. Так, факториал числа вычисляется итеративным способом.

Сравним итеративное решение с рекурсивным:

```
def factorial_recursive(n):
    if n < 0:
        return "Недопустимый ввод! Факториал для отрицательных значений
          не определен."
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

print(factorial_recursive(5)) # Выведет: 120
```

Обе функции дают один и тот же результат для одного и того же входного значения, но используют совершенно разные подходы к решению задачи.

Рекурсия — прием, помогающий писать более чистый и простой код. Как вы уже знаете, она предполагает вызов функции самой себя до тех пор, пока не выполнится определенное условие. Однако, несмотря на привлекательность с точки зрения удобочитаемости и компактности кода, рекурсия может привести к неэффективному решению и ошибкам переполнения стека при обработке больших объемов входных данных.

Итеративное решение, напротив, предполагает использование циклов для повторного выполнения набора инструкций, пока не выполнится определенное условие. Код в этом случае может получиться менее понятным, зато более эффективным, чем рекурсия, особенно при обработке больших объемов входных данных.

Программист должен уметь выбирать наиболее подходящее решение для каждой задачи, поскольку не все задачи лучше решать рекурсивно и не все — итеративно. Иногда оптимальное решение можно получить путем сочетания обоих методов. Поэтому важно понимать плюсы и минусы обоих подходов и уметь переключаться между ними, чтобы писать качественный и эффективный код. Этот навык очень важен при разработке программного обеспечения.

9.2.2. Оптимизация хвостовой рекурсии

Некоторые языки и компиляторы, такие как Scheme и GCC, могут оптимизировать определенные типы рекурсии, в частности хвостовую. Как вы помните, она называется таковой, если рекурсивный вызов является последней операцией в функции. В таком случае нет необходимости добавлять новый кадр стека для каждого вызова. Вместо этого компилятор или интерпретатор может «повторно использовать» кадр стека текущей функции

для следующего вызова. Этот процесс известен как оптимизация хвостового вызова (Tail Call Optimization, TCO).

TCO имеет и другие преимущества, такие как уменьшение объема памяти, используемой программой, и увеличение скорости ее выполнения. Если функция использует обычную рекурсию, то для каждого рекурсивного вызова создается новый кадр стека. Поэтому для размещения кадров стека может потребоваться значительный объем памяти, что чревато возникновением ошибки переполнения стека.

Однако благодаря TCO рекурсивные функции можно использовать, не опасаясь переполнения стека или нерационального расходования памяти. Программисты могут писать код в рекурсивном стиле, который часто легче читать и писать, чем итеративные решения, а TCO позволит иметь производительность, сопоставимую с производительностью итеративного решения, и сохранить элегантность рекурсивного решения. Как следствие, TCO считается важным методом оптимизации языков программирования и компиляторов.

Вот как можно написать на Python функцию `factorial` с хвостовой рекурсией. Имейте в виду, что в действительности этот язык не поддерживает TCO, но если бы поддерживал, то данная функция получила бы неплохое преимущество:

```
def factorial_tail_recursive(n, accumulator=1):
    if n < 0:
        return "Недопустимый ввод! Факториал для отрицательных значений
          не определен."
    elif n == 0 or n == 1:
        return accumulator
    else:
        return factorial_tail_recursive(n - 1, n * accumulator)

print(factorial_tail_recursive(5)) # Выведет: 120
```

Эта версия имеет дополнительный параметр `accumulator`, используемый для хранения промежуточных результатов вычислений. Ключевым моментом здесь является то, что рекурсивный вызов — последняя операция, выполняемая в функции, то есть после нее не производится никаких вычислений. Это отличительная черта хвостовой рекурсии.

Как уже было сказано выше, язык Python не поддерживает TCO, так что эта версия не будет работать более эффективно, чем обычная рекурсивная реализация. Но в языках, поддерживающих ее, этот трюк может принести немало пользы!

На этом мы завершаем краткий обзор итеративных подходов и приемов, лежащих на границе между рекурсивными и итеративными методами. По мере их изучения вы обнаружите, что владение ими может помочь вам в решении сложных задач.

Просто помните, что в информатике и программировании не всегда существует единственный «лучший» способ решения задачи. Оба стиля, рекурсивный и итерационный, имеют свои области применения, и хорошие программисты должны свободно владеть обоими. Выбор между рекурсией и итерацией может зависеть от нескольких факторов, таких как конкретные особенности задачи, требования к эффективности, поддерживаемые языком оптимизации и даже личные предпочтения.

Чтобы хорошо усвоить эти концепции, попрактикуйтесь в решении различных задач. Как и при изучении любого нового языка, чем больше вы его используете, тем более естественным становится владение им. Попробуйте решать задачи как итеративно, так и рекурсивно. Это позволит не только закрепить ваши навыки разработки алгоритмов, но и получить более полное понимание плюсов и минусов каждого подхода в разных ситуациях.

В следующих разделах мы перейдем к более сложным методам разработки алгоритмов. Каждый из них может стать еще одним мощным инструментом в вашем арсенале. Но помните: инструмент полезен ровно настолько, насколько умело вы им пользуетесь. Поэтому продолжайте практиковать и применять эти концепции — и непременно станете опытным программистом.

9.3. Поиск с возвратом

Поиск с возвратом — очень полезный алгоритмический метод, позволяющий решать широкий спектр сложных задач за разумное время. Обычно он используется в задачах принятия решений, где набор потенциальных вариантов может быть организован в виде дерева или графа решений.

Основная идея поиска с возвратом заключается в том, чтобы создавать решение постепенно, шаг за шагом, проверяя, улучшает ли каждый шаг общее решение. Если нет, то шаг удаляется и предпринимается попытка перейти к следующему шагу.

Отличным примером задач, которые можно решить с помощью поиска с возвратом, служит головоломка про *N* ферзей. Согласно условиям головоломки,

вы должны разместить N ферзей на шахматной доске $N \times N$ так, чтобы никакие два ферзя не угрожали друг другу. Это означает, что никакие два ферзя не могут находиться в одной строке, столбце или диагонали.

Поиск с возвратом позволяет решать и другие типы задач, такие как поиск кратчайшего пути в лабиринте или выявление лучшей комбинации предметов для упаковки в рюкзак. В целом поиск с возвратом дает возможность систематически и эффективно решать множество разных видов задач.

Вот простое решение на Python:

```
def isSafe(board, row, col, N):
    # Проверить горизонталь влево
    for i in range(col):
        if board[row][i] == 1
            return False

    # Проверить диагональ вверх-влево
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board{i][j] == 1:
            return False

    # Проверить диагональ вниз-влево
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col, N):
    if col >= N:
        return True

    for i in range(N):
        if isSafe(board, i, col, N):
            board[i][col] = 1

            # Рекурсивно расставить остальных ферзей
            if solveNQUtil(board, col + 1, N) == True:
                return True

            # Если размещение ферзя в поле board[i][col] не приводит
            # к решению, то убрать ферзя из поля board[i][col]
            board[i] [col] = 0

    return False

def solveNQ(N):
    board = [[0]*N for _ in range(N)]
```

```
    if solveNQUtil(board, 0, N) == False:
        print("Решения не существует")
        return False

    for i in range(N):
        for j in range(N):
            print (board[i][j], end =" ")
        print()

    return True

# тест
solveNQ(4)
```

Функция `solveNQUtil` в этом примере является рекурсивной и реализует поиск с возвратом. В качестве аргументов она принимает частичное решение (`board`) и номер вертикали (`col`). Функция пробует поместить ферзя в каждое поле текущей вертикали, а затем повторяет попытку для следующей вертикали. Если размещение в текущей вертикали невозможно, то возвращается значение `False`. Если же возможно, то ферзь ставится в выбранное поле и попытка повторяется для следующей вертикали.

Если алгоритму не удается найти поле для установки ферзя в следующей вертикали, то он снимает ферзя с текущего поля и возвращается назад, чтобы попробовать следующую альтернативу. Этот процесс продолжается до тех пор, пока не будет найдено решение или пока не будут исчерпаны все альтернативы. Прелесть поиска с возвратом в том, что он позволяет систематически исследовать различные варианты и исключить те, которые не приводят к положительному результату, тем самым увеличивая шансы найти оптимальное решение при следующих попытках.

Поиск с возвратом используется для решения сложных задач, не имеющих аналитического решения, или при отсутствии других, более доступных методов. По сути, он предполагает, что принимается некое решение, и если оно не приводит к успеху, то отменяется (выполняется возврат). Идея этого способа поиска решения методом проб и ошибок заключается в том, что он позволяет человеку, решающему задачу, исследовать различные альтернативы, которые могут дать нужный результат, отсеивать непродуктивные варианты и таким образом повышать свои шансы найти оптимальное решение.

Поиск с возвратом — интересная концепция, применимая ко множеству различных задач. Помимо обсуждавшейся выше задачи *N* ферзей, с его помощью можно решать такие головоломки, как судоку, поиск кратчайшего пути в лабиринте, перестановка элементов заданной последовательности и многие другие.

Важно понимать, что, несмотря на действенное решение некоторых задач, поиск с возвратом не всегда оказывается оптимальным вариантом. Это происходит в силу самых разных факторов. Так, этот алгоритм может неэффективно решать задачи с большими входными пространствами.

Кроме того, при использовании поиска с возвратом следует тщательно выбирать условие остановки. Алгоритм работает путем рекурсивной проверки решений, поэтому, не определив должным образом момент, когда функция должна остановиться, вы можете попасть в бесконечный цикл или получить неоправданно долгие вычисления.

Поиск с возвратом играет крайне важную роль в арсенале разработчика алгоритмов, но, как и любой другой инструмент, его следует использовать по назначению и с осторожностью. Освоение этого алгоритма требует большого практического опыта по решению задач, чем я вам и рекомендую заняться.

Помните: красота поиска с возвратом заключается в его простоте и элегантности. Все дело в выборе, и если он не приводит к решению, то вы можете просто отступить и попробовать сделать другой. Этот метод проб и ошибок может стать отличной стратегией решения сложных задач, и навык владения им определенно стоит развивать.

9.4. Метод ветвей и границ

Метод ветвей и границ — эффективный и один из наиболее популярных методов информатики, используемых для решения сложных задач комбинаторной оптимизации. Задачи такого типа распространены повсеместно и встречаются во многих областях, таких как логистика, производство, транспортировка и планирование. Для них характерно наличие конечного множества возможных решений, среди которых необходимо найти лучшее.

Алгоритм ветвей и границ особенно эффективен при решении задач такого типа, поскольку он стратегически перемещается по пространству потенциальных решений, отбрасывая по пути неоптимальные решения и избавляя от необходимости перечислять и оценивать все возможности. Этот подход особенно полезен, когда количество возможных решений очень велико, как часто бывает во многих реальных приложениях.

Основная идея алгоритма ветвей и границ заключается в разделении задачи на более мелкие подзадачи и рекурсивном решении каждой из них эффективным способом. Затем, используя функцию границ для отбрасывания подзадач, которые не могут привести к оптимальному решению, алгоритм

еще больше сокращает пространство поиска. Итеративно применяя эти шаги, алгоритм в итоге приходит к оптимальному решению, избегая при этом ненужных вычислений, которые потребовались бы при использовании других методов.

9.4.1. Принцип работы метода ветвей и границ

Как следует из его названия, метод основан на двух ключевых операциях.

1. **Ветвление.** Это широко используемый способ решения задач, заключающийся в разделении сложной задачи на более мелкие и управляемые подзадачи. Каждая из них соответствует ветви в дереве решений, что позволяет использовать более организованный и систематический подход к поиску решения. При разбиении задачи на более мелкие части легче выявлять ключевые проблемы и определять целевые решения. Это не только экономит время, но и гарантирует, что все аспекты задачи будут тщательно проанализированы и обработаны. В целом ветвление — эффективная стратегия решения задач, которая может значительно повысить эффективность процессов принятия решений.
2. **Ограничение.** На данном этапе вычисляются нижняя и верхняя границы решения для каждой подзадачи. Имея эти границы, можно более четко видеть, какие варианты доступны для каждой подзадачи, и лучше понимать, какие ветви следует исследовать, а какие отсеять. Так, если при поиске минимального решения обнаруживается, что нижняя граница подзадачи выше текущего лучшего решения, то можно с уверенностью отсеять эту подзадачу и не рассматривать ее дальше. Этот метод помогает эффективнее справляться с задачами и в итоге может привести к оптимальным решениям.

Обсудим этот метод на примере одной из классических задач комбинаторной оптимизации.

9.4.2. Задача о коммивояжере

Задача поиска кратчайшего маршрута, который проходит через все города ровно один раз и возвращается в начальную точку, для данного списка городов и расстояний между ними — интересная и сложная. Речь идет о поиске не просто произвольного маршрута, пересекающего все города, а самого эффективного.

Это классическая задача информатики, которую часто называют задачей о коммивояжере. Она применяется во множестве реальных ситуаций: от оптимизации маршрутов доставки до минимизации транспортных расходов продавца, посещающего нескольких клиентов в разных городах. Один из подходов к решению этой задачи заключается в использовании таких алгоритмов, как алгоритм ближайшего соседа или алгоритм 2-Opt, отыскивающих оптимальное решение за разумное время.

Однако найти действительно оптимальное решение часто невозможно, поэтому вместо него используются приближенные. Несмотря на сложность, задача о коммивояжере продолжает оставаться активной областью исследований, при этом постоянно разрабатываются новые алгоритмы и методы поиска лучших и более эффективных решений.

```
# Это простое представление задачи о коммивояжере
# Каждый кортеж представляет город, а второй элемент в этих кортежах
# определяет расстояние от начальной точки маршрута
cities = [(1, 10), (2, 15), (3, 20), (4, 30)]
```

Простейшим решением было бы сгенерировать все возможные сочетания городов, вычислить стоимость каждого сочетания, а затем выбрать вариант с наименьшей стоимостью. Но этот подход не годится для больших исходных данных из-за факториальной временной сложности.

Однако решение методом ветвей и границ может значительно сократить пространство поиска. Стратегия состоит в том, чтобы создать приоритетную очередь путей. Изначально очередь состоит из одного элемента — пути, который содержит только начальный город.

Затем делается следующее.

1. Из очереди удаляется путь с наименьшей стоимостью.
2. Если этот путь пересекает каждый город, то принимается в качестве нового решения при условии, что его стоимость меньше стоимости текущего решения.
3. В противном случае для каждого города, который путь еще не пересекает, создается новый путь, который расширяет текущий, и в него добавляется этот город. Каждый из этих новых путей добавляется в приоритетную очередь.

Этот процесс можно оптимизировать, вычисляя нижние границы путей и используя их в качестве стоимости в приоритетной очереди. Нижняя граница

пути — это сумма стоимости текущего пути и минимальной стоимости соединения последнего города на пути с не входящими в него городами.

Вот псевдокод, описывающий этот алгоритм:

```
function TSP(cities):
    Create a min heap 'pq' and insert the origin city
    while pq is not empty
        path = pq.extract_min()
        If path includes all sities and cost of path < cost
          of current min_cost_path:
            min_cost_path = path
        Else:
            For each city 'c' not in path:
                new_path = path + c
                If cost(new_path) < cost(min_cost_path):
                    pq.insert(new_path)
    Return min_cost_path
```

Этот алгоритм значительно сокращает пространство поиска за счет разумного усечения дерева решений, что является сутью метода ветвей и границ. Однако важно помнить, что эффективность данного метода сильно зависит от качества используемых границ. Хорошая граница может помочь существенно сократить пространство поиска, а плохая — привести к незначительному сокращению, вынуждая алгоритм выполнить обход всех возможных решений.

В этом заключается суть метода ветвей и границ. В сочетании с хорошими стратегиями ограничения он может помочь относительно легко решать сложные задачи оптимизации. Продуманно перемещаясь по пространству решений, вы можете сосредоточиться на лучшем решении, не теряясь в лабиринте возможностей.

9.4.3. Сложность и практическое использование

Метод ветвей и границ используется для решения задач недетерминированной полиномиальной сложности (NP-трудных — NP-hard problem), которые, как известно, не имеют эффективного решения. Как и в случае поиска с возвратом, оценить временную сложность этого метода довольно трудно. Но при худшем развитии событий он может потребовать экспоненциального времени.

Однако на практике метод ветвей и границ часто оказывается более эффективным, чем другие методы, такие как простой перебор. Это связано с использованием стратегии усечения, помогающей сберечь время. Количество сэкономленного времени в конечном счете зависит от эффективности вычисления границ и от того, насколько сокращается пространство поиска.

При худшем развитии событий метод ветвей и границ может потребовать много времени и тем не менее считается более эффективным, чем другие простейшие методы, благодаря способности быстрее и лучше находить оптимальное решение. Поэтому, даже несмотря на то, что он может занять больше времени, этот метод часто является лучшим вариантом, когда речь идет о решении задач недетерминированной полиномиальной сложности.

Помимо задачи о коммивояжере, метод ветвей и границ используется и в других областях.

1. **Целочисленное программирование.** Это универсальный метод оптимизации, имеющий широкий спектр практического применения. Он используется для решения сложных задач, в которых какие-то из неизвестных переменных должны быть целыми числами. Среди примеров, когда целочисленное программирование полезно, можно назвать планирование, составление графиков и многое другое.

   Например, этот метод позволяет решить задачу распределения рабочих на заводе и гарантировать, что в каждую смену будет назначено нужное количество работников. Кроме того, целочисленное программирование может помочь в составлении бюджета капиталовложений, устанавливая наилучшее распределение средств для различных проектов с учетом таких ограничений, как финансовые лимиты и сроки проекта.

   В области исследования операций целочисленное программирование играет крайне важную роль в решении задач во многих отраслях и продолжает оставаться областью активных исследований и разработок.

2. **Задача о рюкзаке 0/1.** Это хорошо известная задача оптимизации, которая часто используется для иллюстрации возможностей динамического или целочисленного программирования. Она заключается в выборе подмножества предметов, которые поместятся в рюкзак с ограниченной грузоподъемностью. Цель состоит в том, чтобы максимизировать общую ценность выбранных предметов.

Эта задача играет значимую роль во многих областях, например в информатике, исследовании операций и инженерии. Ее часто используют для моделирования реальных задач, таких как распределение ресурсов и управление проектами. Задача о рюкзаке 0/1 тщательно изучена, и для ее решения разработано множество алгоритмов, в том числе алгоритмы динамического программирования, метод ветвей и границ и генетические алгоритмы.

Несмотря на сложность, задача часто применяется на практике и является важной темой в области оптимизации.

3. **Задача распределения заданий.** Эта задача играет важную роль в различных областях, таких как логистика, управление цепочками поставок и транспорт. В ней *n* заданий нужно распределить между *n* работниками так, чтобы минимизировать общую стоимость их выполнения. Задачу можно решить с помощью метода ветвей и границ. Он предполагает разделение задачи на более мелкие подзадачи, поиск оптимального решения для каждой из них и их последующее объединение для поиска глобального оптимума. Метод известен своей эффективностью и оперативностью при решении оптимизационных задач и широко используется в различных приложениях.

   Помимо метода ветвей и границ, для решения этой задачи можно использовать и другие методы: линейное программирование, динамическое программирование и эвристические алгоритмы. Эти методы имеют свои достоинства и недостатки, и выбор какого-то из них зависит от сложности, размера и конкретных требований задачи.

   Например, линейное программирование подходит для решения крупномасштабных задач с линейными ограничениями, а эвристические алгоритмы — для решения сложных и нелинейных задач, которые невозможно решить оптимально с помощью аналитических методов. В целом задача распределения заданий является важной для оптимизации, и для ее эффективного решения можно использовать множество методов.

4. **ИИ и машинное обучение.** Метод ветвей и границ также часто используется в различных областях искусственного интеллекта и машинного обучения. В ИИ он служит для принятия решений, особенно когда существует большое количество возможных вариантов выбора и становится сложно изучить все возможные альтернативы. Используя метод ветвей и границ, системы искусственного интеллекта могут эффективно исследовать различные варианты и определять лучшую альтернативу.

> В машинном обучении метод ветвей и границ используется для выбора признаков — важного процесса, в который входит определение наиболее информативных признаков из набора доступных. Применяя алгоритмы ветвей и границ, модели машинного обучения могут исследовать пространство доступных признаков и определить наиболее релевантные. Это помогает повысить точность и производительность моделей, увеличивая их эффективность в части прогнозирования результатов и выявления закономерностей в данных.

Помните, что эффективность метода ветвей и границ во многом зависит от качества границ и скорости их вычисления. Чем точнее вычисляются границы, тем более широкое пространство поиска можно исключить и, следовательно, тем быстрее будет найдено оптимальное решение.

Как мы уже говорили, метод ветвей и границ — мощный инструмент для решения сложных задач. Однако его эффективность напрямую связана с качеством используемой функции ограничения. Плохая функция может привести к незначительному сокращению пространства вариантов на каждом шаге и вынудить алгоритм работать не лучше, а потенциально хуже, чем метод простого перебора.

Кроме того, метод ветвей и границ может потребовать большого объема памяти. Ему часто приходится хранить в памяти большие фрагменты дерева поиска, что может быть проблематично при решении задач с обширным пространством поиска.

Стратегия обхода дерева поиска тоже может оказать большое влияние на производительность алгоритма. Поиск в глубину обычно применяется из-за того, что эффективно использует память, но он не всегда способен быстро найти лучшее решение. В то же время с этим может справиться поиск в ширину, но ему часто требуется куда больший объем памяти.

Из вышесказанного следует, что метод ветвей и границ может быть очень действенным, но не всегда является лучшим вариантом решения конкретной задачи. Эффективное использование метода требует тщательной проработки и изучения рассматриваемой задачи.

Таким образом, несмотря на все достоинства этого метода, его следует применять разумно и учитывать потенциальные последствия. Никогда не забывайте тщательно проанализировать задачу, прежде чем выбирать подход к ее решению!

9.5. Практические задачи

Мы рады предложить несколько задач, которые помогут вам проверить и применить знания, полученные в этой главе.

Задача 1. Рекурсия: числа Фибоначчи

Напишите рекурсивную функцию, которая генерирует n-е число Фибоначчи. Помните, что последовательность Фибоначчи представляет собой ряд чисел, каждое из которых является суммой двух предыдущих, и обычно начинается с 0 и 1.

```
def fibonacci(n):
    if n <= 0:
        return "Ввод должен быть целым положительным числом."
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Задача 2. Итерации: факториал

Напишите итерационную функцию, вычисляющую факториал заданного числа. Факториал неотрицательного целого числа n — это произведение всех натуральных чисел, меньших или равных n.

```
def factorial(n):
    if n < 0:
        return "Ввод должен быть целым неотрицательным числом."
    else:
        result = 1
        for i in range(2, n+1):
            result *= i
        return result
```

Задача 3. Поиск с возвратом: задача N ферзей

Задача N ферзей заключается в том, чтобы разместить N ферзей на шахматной доске $N \times N$ так, чтобы никакие два ферзя не угрожали друг другу, то есть чтобы никакие два ферзя не находились на одной горизонтали, вертикали или диагонали.

```
def solveNQueens(n):
    def can_place(pos, occupied_positions):
        for i in range(len(occupied_positions)):
            if occupied_positions[i] == pos or \\
                occupied_positions[i] - i == pos - len(occupied_positions) or \\
                occupied_positions[i] + i == pos + len(occupied_positions):
                return False
        return True

    def place_queens(n, index, occupied_positions):
        if index == n:
            return [occupied_positions]
        else:
            result = []
            for pos in range(n):
                if can_place(pos, occupied_positions):
                    result += place_queens(n, index + 1, occupied_positions + [pos]
            return result

    result = place_queens(n, 0, [])
    return [["." * i + "Q" + "." * (n - i -1) for i in pos] for pos in result]
```

Задача 4. Ветви и границы: задача о коммивояжере

Задача о коммивояжере звучит так: «при наличии списка городов и расстояний между ними определить кратчайший возможный маршрут, пересекающий каждый город ровно один раз и возвращающийся в исходную точку».

Задача о коммивояжере — это NP-трудная задача комбинаторной оптимизации, занимающая важное место в исследовании операций и теоретической информатике. При большом количестве входных данных трудно найти оптимальное решение, используя простой перебор вариантов, поэтому для ее решения рекомендуется применять метод ветвей и границ.

Надо отметить, что написать эффективный код решения задачи о коммивояжере с помощью метода ветвей и границ довольно сложно, и, вероятно, разработка такого кода — не лучшее упражнение, с которого стоит начинать изучение этого метода. Поэтому советуем внимательно ознакомиться с теоретическими аспектами этой задачи и попытаться понять, как метод ветвей и границ может помочь сократить пространство поиска.

Задачи в этом разделе призваны помочь вам понять изученный материал. Попробуйте решить их, и если столкнетесь с какой-либо проблемой, то поищите возможные решения и объяснения. Ваша цель — учиться, а этот процесс требует времени и терпения. Не торопитесь, наслаждайтесь решением этих задач. Успехов!

Резюме

В данной главе мы рассмотрели множество способов проектирования алгоритмов, предназначенных для решения вычислительных задач. Эти методы лежат в основе решения задач в информатике, и, поняв их, вы сможете эффективно справляться со многими сложными проблемами.

Мы начали с рекурсии — метода, применяемого в случаях, когда решение задачи зависит от решений более мелких ее экземпляров. Он предполагает вызов функции с меньшей задачей, решение которой будет способствовать решению исходной задачи. Мы рассмотрели основные понятия, такие как базовый и рекурсивный случаи, которые помогают гарантировать, что рекурсия не будет продолжаться бесконечно. Мы также изучили некоторые ключевые проблемы, связанные с использованием рекурсии: переполнение стека и необходимость применения мемоизации во избежание избыточных вычислений.

Далее мы исследовали итеративный подход, который, в отличие от рекурсии, использует конструкцию цикла для многократного решения небольших частей задачи, пока не будет получено полное решение. Итеративные алгоритмы часто имеют меньшую пространственную сложность, чем рекурсивные, поскольку не нуждаются в дополнительном пространстве для хранения рекурсивных вызовов на стеке. Однако выбор между итеративным и рекурсивным подходами часто зависит от решаемой задачи, а также требований к ясности и эффективности решения.

Затем мы перешли к поиску с возвратом — алгоритмическому методу рекурсивного решения задач — и попытались построить решение постепенно, делая по одному шагу за раз. Если решение неосуществимо или не приводит к полному решению, то алгоритм отменяет последний шаг (делает возврат) и пробует другой вариант. Такой подход особенно эффективен для задач, в решение которых входит просмотр последовательности вариантов, как, например, в классической задаче о N ферзях.

Далее мы рассмотрели метод ветвей и границ — мощную стратегию, используемую для решения задач оптимизации. Этот метод делит задачу на подзадачи (ветви) и вычисляет оптимистическую оценку лучшего решения, которое может быть получено. Если граница подзадачи не лучше оптимальной из найденных к текущему моменту, то мы можем отбросить эту подзадачу, не исследуя ее дальше. Мы кратко обсудили задачу о коммивояжере как пример применения метода ветвей и границ.

В конце главы вам были предложены практические задачи, связанные с этими методами и позволяющие закрепить полученные знания. Благодаря решению этих задач вы сможете отточить навыки применения данных методов на практике и лучше понять, как и когда использовать каждый из них.

Итак, методы проектирования алгоритмов помогают решать задачи информатики структурированно. Часто при первом знакомстве с задачей ее сложность может ошеломить. Но прелесть в том, что большие задачи можно разбить на более мелкие и управляемые и создавать решение постепенно. И помните: освоение описанных методов — это путешествие, полное увлекательных испытаний и важных открытий. Поэтому продолжайте практиковаться, и успехов вам!

Глава 10
ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ АЛГОРИТМОВ

В этой главе мы углубимся в практику применения теории алгоритмов, методов проектирования и структур данных, которые изучали на протяжении книги. Мы постараемся продемонстрировать полезность и значимость алгоритмов в различных областях.

По мере продвижения вперед мы расскажем, какую роль играют алгоритмы в таких областях, как базы данных, искусственный интеллект, машинное обучение, сетевая маршрутизация, криптография и многие другие. В каждом разделе мы представим практические примеры, демонстрирующие конкретные алгоритмы, которые используются для эффективного решения задач или повышения производительности.

Первым делом мы сосредоточимся на алгоритмах в базах данных и посмотрим, как оптимизировать хранение и поиск информации, что позволит обеспечить эффективное управление большими наборами данных. Для этого мы познакомимся с несколькими алгоритмическими методами, такими как индексирование, сортировка и поиск, а также рассмотрим их применение в системах управления базами данных.

Мы надеемся, что благодаря этому исследованию случаев реального применения алгоритмов вы получите более полное представление об их огромном потенциале и вдохновитесь на дальнейшее изучение их возможностей.

10.1. Алгоритмы в базах данных

Базы данных являются важнейшим компонентом практически всех современных отраслей и с помощью широкого спектра алгоритмов позволяют эффективно хранить данные, выполнять их поиск и обработку. Эти алгоритмы работают в тесной связи, обеспечивая бесперебойное и точное функционирование баз данных.

Один из основных алгоритмов, используемых базами данных, — индексирование. Представьте библиотеку с тысячами книг, но без системы каталогизации. В такой библиотеке было бы невероятно сложно найти конкретную книгу, не так ли? Но если книги упорядочить, например, по именам авторов, то найти нужную книгу можно гораздо быстрее. Индексирование в базах данных служит именно этой цели — упорядочению данных в удобной для поиска форме.

В базах данных используется несколько распространенных алгоритмов индексации, в том числе алгоритм B-деревьев. Как вы помните из главы 8, это самобалансирующиеся деревья поиска, лучше всего подходящие для приложений, преимущественно читающих данные. Они гарантируют доступ к данным, имеющим логарифмическую временную сложность, что делает их идеальными для применения в базах данных, которым необходимо поддерживать быстрый поиск записей. Алгоритм B-дерева постоянно балансирует дерево по мере вставки новых или удаления старых ключей, гарантируя его оптимальность для операций чтения.

Помимо индексации, базы данных используют несколько других алгоритмов, обеспечивающих правильное их функционирование. Например, базы применяют алгоритмы обработки запросов для получения данных, исходя из критериев в запросах пользователей. Кроме того, с помощью алгоритмов поддерживается согласованность данных даже в распределенных системах. Все вместе эти алгоритмы обеспечивают эффективное и надежное функционирование баз данных, которые необходимы современным отраслям.

Пример

Рассмотрим пример индексации B-дерева. Имейте в виду, что приведенный ниже код — лишь упрощенное представление, фактическая реализация может быть более сложной:

```
# Создание узла
class BTreeNode:
    def __init__(self, leaf=False):
        self.leaf = leaf
        self.keys = []
        self.child = []

# B-дерево
class BTree:
    def __init__(self, t):
        self.root = BTreeNode(True)
```

```
    # Вставка узла
    def insert(self, k):
        root = self.root
        if len(root.keys) == (2*t) - 1:
            temp = BTreeNode()
            self.root = temp
            temp.child.insert(0, root)
            self.split_child(temp, 0)
            self.insert_non_full(temp, t)
        else:
            self.insert_non_full(root, k)

# Другие методы для поддержки операций разбиения и вставки...
```

Запросы в базах данных — важнейшая область, в которой алгоритмы играют значительную роль. SQL-запросы, которые мы посылаем базам данных, оптимизируются с помощью различных алгоритмов, определяющих наиболее эффективный способ соединения двух таблиц на основе представленных критериев. К числу этих алгоритмов относятся алгоритмы соединения вложенными циклами, соединения сортировкой слиянием и хеш-соединения.

Так, алгоритм соединения вложенными циклами сравнивает две таблицы, перебирая строки одной таблицы и путем сканирования другой таблицы проверяя, удовлетворяет ли каждая строка условию соединения. Алгоритм сортировкой слиянием сортирует обе таблицы на основе условия соединения, а затем объединяет их и формирует окончательный набор результатов. Алгоритм хеш-соединения создает хеш-таблицу для одной таблицы, а затем сравнивает строки другой таблицы с уже созданной, чтобы найти совпадающие пары.

Таким образом, алгоритмы играют решающую роль в индексировании, обработке сложных взаимосвязей между таблицами и в других операциях с базами данных. Понимая и применяя эти алгоритмы, мы можем создавать более эффективные системы баз данных. Это, в свою очередь, приводит к разработке более быстрых и надежных приложений. Поэтому всегда полезно тщательно изучить алгоритмы баз данных, чтобы полностью раскрыть их потенциал!

Важно коснуться темы транзакций в базах данных и алгоритмов управления параллелизмом. Они позволяют поддерживать свойства ACID (Atomicity, Consistency, Isolation, Durability — атомарность, согласованность, изоляция, долговечность) баз данных.

10.1.1. Двухфазная блокировка (2PL)

Двухфазная блокировка (Two-Phase Locking, 2PL) — широко используемый метод управления параллелизмом в системах баз данных. Он помогает обеспечить упорядоченность — ключевое свойство обработки транзакций, когда они выполняются последовательно.

Метод 2PL состоит из двух основных этапов: блокировки и разблокировки. На первом этапе транзакция получает все блокировки, которые ей нужны для выполнения своих операций. Блокировки гарантируют, что никакая другая транзакция не изменит данные, с которыми работает текущая транзакция, и это предотвращает любые конфликты. На втором этапе освобождаются все блокировки, установленные на первом. После этого их больше нельзя получить, что гарантирует выполнение транзакций в строгой последовательности.

Метод 2PL дает системам баз данных несколько преимуществ: помогает соблюсти согласованность данных, предотвращая одновременное изменение несколькими транзакциями, и обеспечивает высокую степень параллелизма, позволяя одновременно выполнять несколько транзакций и гарантируя при этом целостность данных. Надо отметить, что метод 2PL является эффективным средством обеспечения корректности и согласованности транзакций в системах баз данных.

10.1.2. Управление параллельным доступом с помощью многоверсионности

Управление параллельным доступом с помощью многоверсионности (Multiversion Concurrency Control, MVCC) — это алгоритм, позволяющий нескольким транзакциям обращаться к одним и тем же данным без конфликтов. Для этого он создает новую версию объекта базы каждый раз, когда выполняется запись, что дает параллельным транзакциям возможность работать с отдельными версиями одной и той же записи. Этот метод широко используется в PostgreSQL и MySQL (InnoDB).

Реализация MVCC особенно полезна в ситуациях, когда нескольким пользователям или приложениям требуется одновременный доступ к одним и тем же данным. Например, представьте, что два пользователя пытаются одновременно обновить одну и ту же запись в базе данных. Без MVCC одна из транзакций будет заблокирована до завершения другой. Это может привести к снижению производительности и даже к нарушению согласованности данных.

Однако благодаря MVCC обе транзакции могут выполняться независимо, поскольку работают с разными версиями одной и той же записи. Это означает, что данные можно обновлять и читать одновременно без каких-либо конфликтов. Кроме того, поскольку сохраняется каждая версия записи, можно получить доступ к историческому представлению данных, что впоследствии может пригодиться для аудита или анализа тенденций.

В целом MVCC — мощный алгоритм, помогающий значительно повысить производительность и надежность приложений, которым требуется одновременный доступ к данным. Позволяя нескольким транзакциям работать с отдельными версиями одной и той же записи, MVCC предоставляет гибкое и масштабируемое решение проблемы управления параллелизмом в современных системах баз данных.

Наконец, не будем забывать об алгоритмах восстановления баз данных, таких как ARIES (Algorithm for Recovery and Isolation Exploiting Semantics — алгоритм восстановления и изоляции с использованием семантики), которые гарантируют возможность восстановления базы данных после сбоев, при этом свойства ACID сохраняются. С помощью таких методов, как журналирование и контрольные точки, эти алгоритмы отслеживают изменения и отменяют их или используют для обеспечения согласованности.

Помните, что изучать эти алгоритмы необходимо не только ради их реализации — в конце концов, они уже работают в системах баз данных, которые мы используем! Понимая эти алгоритмы, вы сможете выбрать базу данных, подходящую под ваши условия, а также получить ценную информацию при отладке проблем или аномалий производительности.

Подводя итог, можно сказать, что базы данных — прекрасный пример реального использования алгоритмов. Они дают отличную возможность увидеть, как теории и концепции, которые мы изучали, можно объединять для решения практических задач. От индексации до обработки запросов, от управления транзакциями до восстановления после сбоев — в основе всего этого лежат алгоритмы!

10.2. Алгоритмы в искусственном интеллекте

Искусственный интеллект (ИИ) — постоянно расширяющаяся область, в которой каждый день происходят новые достижения. ИИ применяется повсюду: в беспилотных автомобилях и распознавании речи, в системах рекомендаций и диагностике заболеваний. Однако базируется ИИ на алгоритмах.

Благодаря сложным математическим моделям машины могут изучать данные, принимать решения и имитировать человеческий интеллект, и эти модели продолжают развиваться и совершенствоваться по мере проведения новых исследований. Объем данных, генерируемых человечеством каждый день, растет. Параллельно увеличивается и потребность в эффективных алгоритмах. Все это делает ИИ важнейшей областью познания как для исследователей, так и для практиков.

10.2.1. Алгоритмы машинного обучения

Машинное обучение (МО) как ветвь искусственного интеллекта — постоянно развивающаяся область, цель которой — наделить машины способностью изучать данные, выявлять закономерности и делать прогнозы или принимать решения при минимальном вмешательстве человека. В связи с тем, что во всех отраслях все больше внимания уделяется автоматизации и эффективности, МО стали активно использовать на предприятиях, стремящихся оптимизировать свою деятельность.

Алгоритмы, применяемые в машинном обучении, позволяют компьютерам обрабатывать огромные объемы данных, выявлять тенденции и изучать прошлый опыт в целях повышения производительности в будущем. Фактически МО уже произвело революцию во многих сферах нашей жизни: от персонализированных рекомендаций на Netflix и Amazon до беспилотных автомобилей и обнаружения мошенничества в банковской сфере. Эта область продолжает развиваться, поэтому в ближайшие годы мы можем ожидать появления еще более инновационных способов ее применения.

Рассмотрим несколько примечательных примеров.

Линейная регрессия

Линейная регрессия — это статистический метод, обычно используемый для прогнозирования непрерывной выходной переменной на основе одного или нескольких входных признаков, и форма обучения с учителем, в которой предполагается линейная связь между входными признаками и выходными прогнозами. В модели линейной регрессии выходная переменная моделируется как линейная комбинация входных переменных. Этот подход широко используется в различных областях, таких как экономика, бизнес, здравоохранение и многие другие.

Линейную регрессию можно использовать для решения самых разных задач: прогнозирования цен на дома на основе их размеров и местоположения,

прогнозирования продаж продукта или оценки риска развития определенного заболевания на основе набора рисков. Появление машинного обучения и больших данных превратило линейную регрессию в еще более эффективный инструмент прогнозирования и анализа.

Стоит отметить, что линейная регрессия предполагает линейную связь между входными и выходными переменными, но в реальных сценариях это не всегда так. В них можно с успехом применять более сложные модели, такие как полиномиальная или нелинейная регрессия. Однако линейная регрессия остается ценным и широко используемым средством анализа и прогнозирования данных.

Рассмотрим простой пример реализации линейной регрессии на Python:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
import pandas as pd

# Загрузить набор данных
url = "<https://raw.githubusercontent.com/AdiPersonalWorks/Random/master/
student_scores%20-%2@student_scores.csv>"
dataset = pd.read_csv(url)

# Подготовить данные
X = dataset.iloc[{:, :-1].values
y = dataset.iloc[:, 1].values

# Разбить данные
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)

# Обучить алгоритм
regressor = LinearRegression()
regressor.fit(X_train.reshape(-1,1), y_train)

# Получить прогноз
y_pred = regressor.predict(X_test)
```

Код в этом примере загружает набор данных, подготавливает его, делит на обучающую и контрольную выборки, а затем обучает модель линейной регрессии. В заключение полученная модель используется для прогнозирования контрольных данных. Это типичная последовательность действий в машинном обучении и практический пример использования алгоритмов ИИ.

Однако все это — лишь малая толика информации! Алгоритмы искусственного интеллекта — обширная и интересная область с огромными

возможностями, и в ней всегда есть что изучать и исследовать. Поэтому погружайтесь в нее и получайте удовольствие от открытий!

Деревья решений

Это еще один алгоритм обучения с учителем, широко используемый в машинном обучении. Графически его можно представить в виде дерева всех возможных решений, основанных на определенных условиях и соответствующих им результатах.

Деревья решений можно использовать как для классификации, так и для решения задач регрессии. Они могут обрабатывать как дискретные, так и непрерывные входные и выходные переменные. Деревья решений легко понять, поэтому их часто выбирают специалисты по данным и машинному обучению. Кроме того, деревья решений можно использовать в сочетании с другими алгоритмами, чтобы создавать сложные модели, которые могут дать более высокие результаты.

Нейронные сети

Это набор алгоритмов, которые действуют по образу и подобию человеческого мозга и состоят из слоев взаимосвязанных узлов, выполняющих сложные вычисления. Они предназначены для выявления закономерностей и толкования данных путем маркировки или кластеризации входных данных.

Узлы нейронной сети образуют слои, каждый из которых обрабатывает выходные данные предыдущего слоя. Входной слой получает исходные данные, а выходной выводит окончательный результат. Между ними может находиться несколько скрытых слоев, которые извлекают из входных данных все более абстрактные признаки.

Нейронные сети успешно применяются в широком спектре приложений: от распознавания изображений и речи до обработки естественного языка и автоматического управления автомобилем. Продолжаются исследования по улучшению их производительности и интерпретируемости, а также по разработке новых архитектур, способных решать более сложные задачи.

О нейросетях мы еще поговорим, когда будем обсуждать глубокое обучение.

Алгоритмы поиска

В ИИ алгоритмы поиска служат для навигации по пространству поиска (графу или древовидному представлению задачи) решения. Существует множество алгоритмов поиска, каждый из которых имеет сильные и слабые

стороны. Например, уже знакомый вам алгоритм поиска A* используется при наличии эвристической функции, а поиск в глубину лучше подходит для обхода больших деревьев. Помимо этих двух алгоритмов, в ИИ используются и другие известные алгоритмы поиска, такие как поиск в ширину, поиск по критерию стоимости и поиск по наилучшему соответствию.

Стоит отметить, что алгоритмы поиска используются не только в ИИ, но и в других областях, таких как информатика, математика и исследование операций. В информатике алгоритмы поиска служат для решения таких задач, как поиск определенного значения в списке или сортировка массива. В математике с помощью алгоритмов поиска можно решать задачи оптимизации, цель которых — найти лучшее решение среди множества возможных. В исследованиях операций алгоритмы поиска применяются в таких задачах, как поиск кратчайшего пути между двумя точками или планирование заданий в целях минимизации затрат.

Таким образом, алгоритмы поиска используются в самых разных областях для решения разнообразных задач. Понимая сильные и слабые стороны различных алгоритмов поиска, можно выбрать правильный и находить решения быстрее и эффективнее.

Алгоритмы оптимизации

Искусственный интеллект часто связан с решением сложных задач, имеющих слишком много потенциальных решений, чтобы их можно было решить методом простого перебора. Алгоритмы оптимизации позволяют отыскать оптимальные решения среди множества возможностей.

В числе алгоритмов оптимизации, обычно используемых в ИИ, можно назвать, например, имитацию отжига, генетические алгоритмы и оптимизацию роем частиц. Эти алгоритмы выполняют поиск в больших наборах потенциальных решений, оценивают каждое из них, сравнивают их между собой и выявляют лучший вариант.

Используя эти методы оптимизации, исследователи и разработчики ИИ могут создавать все более мощные системы, способные решать самые сложные задачи.

Глубокое обучение

Эта область машинного обучения образована набором алгоритмов, моделирующих работу человеческого мозга и называемых искусственными нейронными сетями. Эта разновидность машинного обучения дает компьютерам

возможность учиться без явного программирования. Искусственный интеллект может анализировать огромные объемы сложных данных, поэтому навык глубокого обучения является одним из самых востребованных в данной области.

Глубокое обучение используется также во многих приложениях и сервисах, расширяя возможности автоматизации, и решает задачи высокой сложности, такие как распознавание изображений и речи и обработка естественного языка. Например, глубокое обучение широко применяется в технологиях распознавания лиц, позволяя более точно идентифицировать личность и обеспечить меры безопасности. В здравоохранении глубокое обучение помогает выявлять заболевания по медицинским изображениям, таким как рентгеновские снимки и компьютерная томография.

Один из алгоритмов, широко используемых в глубоком обучении, — это сверточные нейронные сети (Convolutional Neural Networks, CNN), которые обычно применяются для анализа визуальных данных. CNN используются в различных приложениях, в том числе беспилотных автомобилях, где они анализируют изображения с камер в целях обнаружения препятствий и принятия решения об изменении скорости и направления движения. Глубокое обучение имеет безграничный потенциал и может произвести революцию во многих отраслях и улучшить нашу повседневную жизнь.

Вот простой пример модели глубокого обучения на Python, в которой используется библиотека Keras:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# загрузить набор данных
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# разбить на входные (X) и выходные (Y) переменные
X = dataset[:,0:8]
Y = dataset[:,8]

# определить модель keras
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# скомпилировать модель keras
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
# обучить модель keras на наборе данных
model.fit(X, Y, epochs=150, batch_size=10)

# получить классы, предсказанные моделью
predictions = model.predict_classes(X)
```

Код в этом примере обучает простую модель нейронной сети на наборе данных Pima Indians Diabetes, которая поможет предсказать наличие у человека диабета, основываясь на результатах различных диагностических измерений.

В мире ИИ алгоритмы играют ключевую роль, позволяя этим интеллектуальным системам понимать окружающую среду, анализировать ее и учиться на ее основе. Эта интересная область постоянно развивается, делая вероятным появление более сложных и эффективных алгоритмов, которые расширят границы возможностей машин.

10.2.2. Алгоритмы обработки естественного языка в ИИ

Обработка естественного языка (Natural Language Processing, NLP) – интересная область искусственного интеллекта, основное внимание в которой уделяется взаимодействиям компьютеров и людей с помощью естественного языка. Алгоритмы NLP можно использовать для того, чтобы научить компьютеры понимать и даже генерировать человеческую речь. Эта технология может значительно упростить наше общение с машинами и помочь сделать ценные выводы о человеческом языке.

Анализируя большие объемы текстовых данных, алгоритмы NLP могут обучаться распознавать закономерности, определять эмоциональную окраску текста и даже генерировать осмысленные ответы на вопросы пользователя. Достижения в области машинного обучения и технологий обработки естественного языка позволяют ожидать еще более интересных событий в этой области в ближайшие годы.

Вот некоторые примеры приложений NLP:

- перевод текста;
- анализ эмоциональной окраски высказываний;
- распознавание речи;
- извлечение информации;
- обобщение текста;

- чат-боты и виртуальные помощники, такие как Google Assistant, Alexa, Siri и т. д.

Одна из распространенных задач NLP – классификация текста, также известная как анализ эмоциональной окраски. Продемонстрируем пример реализации этого вида анализа на Python с использованием библиотеки NLTK (Natural Language Toolkit):

```
import nltk
from nltk.classify import NaiveBayesClassifier

# набор обучающих данных в формате: оценка, текст
training_data = [
    ('positive', 'I love this sandwich.'),
    ('negative', 'I feel very bad about this.'),
    ('positive', 'This is an amazing place!'),
    ('negative', 'I can\\'t deal with this anymore.'),
    ('positive', 'Everything is wonderful.'),
    ('negative', 'My boss is horrible.')
]

# преобразовать обучающие данные в признаки, с которыми сможет работать NLTK
vocabulary = set(word.lower() for passage in training_data for word
  in nltk.word_tokenize(passage/[1]))
feature_set = [({word: (word in nltk.word_tokenize(x[1])) for word
  in vocabulary}, x{0]) for x in training_data]

# обучить модель
classifier = NaiveBayesClassifier.train(feature_set)

# протестировать модель на некоторых примерах
print(classifier.classify({word: (word in 'I feel amazing'.lower())
  for word in vocabulary}))
print(classifier.classify({word: (word in 'My day was bad'.lower())
  for word in vocabulary}))
```

В этом примере сначала создается словарь на основе обучающих данных. Затем данные преобразуются в набор признаков, с которыми может работать библиотека NLTK. Каждый признак представлен словарем, отображающим слово в логическое значение, которое показывает, содержит ли текст в этом обучающем примере данное слово. Затем проводится обучение классификатора, и в заключение полученный классификатор тестируется на новых входных данных. В роли классификатора в этом примере выступает простой наивный байесовский классификатор, который использует теорему Байеса для классификации текста.

10.2.3. Роль алгоритмов в машинном обучении

Машинное обучение (МО) — интересная и быстро развивающаяся область искусственного интеллекта, которая коренным образом меняет подходы к анализу и интерпретации сложных данных. Используя сложные алгоритмы и статистические модели, МО позволяет выявлять закономерности и особенности, которые иначе могли бы остаться скрытыми. Машинное обучение можно разделить на три основных вида, каждый из которых имеет свои сильные стороны и области применения.

Обучение с учителем — наиболее широко используемый вид МО, когда алгоритм учится прогнозировать выходную переменную на основе входных переменных. Этот вид МО широко используется в таких приложениях, как распознавание изображений и речи, фильтрация спама и анализ эмоциональной окраски высказываний.

Обучение без учителя, напротив, предполагает поиск закономерностей и взаимосвязей в данных при отсутствии конкретной выходной переменной, которую можно было бы предсказать. Этот вид МО используется для решения таких задач, как кластеризация, обнаружение аномалий и уменьшение размерности.

Обучение с подкреплением — это вид МО, в котором алгоритм учится принимать решения на основе вознаграждений и наказаний. Он используется для решения таких задач, как игры и робототехника, где агент должен учиться действовать правильно методом проб и ошибок.

Область МО невероятно интересна, а спектр ее практического применения очень широк, что имеет далекоидущие последствия. Поскольку мы продолжаем разрабатывать более совершенные алгоритмы и модели, потенциал влияния машинного обучения на наш мир поистине безграничен.

Рассмотрим каждый из видов машинного обучения более подробно.

Обучение с учителем

В обучении с учителем алгоритм учится на маркированных данных, где каждая точка данных связана с соответствующей меткой или выходным значением. Алгоритм пытается выявить основную закономерность или взаимосвязь между входными и выходными переменными. Затем он использует полученные знания для прогнозирования новых, ранее неизвестных данных.

Типичным примером алгоритма обучения с учителем может служить алгоритм регрессии, прогнозирующий непрерывный результат. Например, с его помощью можно прогнозировать цены на дома на основе их характеристик, таких как площадь, количество спален и ванных комнат, местоположение и т. д.

В числе других примеров алгоритмов обучения с учителем можно назвать алгоритмы классификации, прогнозирующие категориальный результат, и алгоритмы рекомендаций, прогнозирующие предпочтения пользователя на основе его поведения в прошлом. Обучение с учителем широко используется в различных приложениях, таких как распознавание изображений, распознавание речи, обработка естественного языка и многих других.

Пример

```
from sklearn.linear_model import LinearRegression

X = [[1], [2], (3], [4], [5]] # входные данные
y = [2, 4, 6, 8, 10] # выходные данные

# обучить модель линейной регрессии
model = LinearRegression().fit(x, y)

# предсказать результат для новых входных данных
print(model.predict([[6]])) # Выведет: [12.]
```

Код в этом простом примере обучает модель линейной регрессии из Scikit-Learn, популярной библиотеки машинного обучения на Python. Модель обучается на входных (X) и выходных (y) данных, а затем используется для прогнозирования выходных данных по новым входным данным.

Обучение без учителя

При обучении без учителя алгоритм не получает никаких меток или целевых значений и должен сам выявить закономерности во входных данных. Это означает, что он должен более тщательно работать над выявлением закономерностей и взаимосвязей в данных без всякой помощи со стороны.

Этот вид обучения особенно полезен при работе с большими объемами данных, поскольку позволяет алгоритму выявлять скрытые закономерности. Типичным примером является алгоритм кластеризации, который группирует

данные на основе их сходства. Таким образом он может помочь выявить закономерности и взаимосвязи, неочевидные на первый взгляд. Это может облегчить принятие решений и выявить особенности, которые в противном случае могли бы остаться незамеченными.

Пример

```
from sklearn.cluster import KMeans

X = [[1], [2], [3], [10], [11], [12]] # входные данные

# обучить модель KMeans
model = KMeans(n_clusters=2).fit(X)

# предсказать кластер, к которому относится новое входное значение
print(model.predict([[6]])) # выведет: [0] или [1]
```

Код в этом примере сначала использует модель KMeans из Scikit-Learn для кластеризации входных данных в два кластера, а затем прогнозирует кластер для нового входного значения.

Обучение с подкреплением

Обучение с подкреплением — это разновидность машинного обучения, которая позволяет агенту учиться путем взаимодействия с окружающей средой. Агент получает обратную связь в виде вознаграждений или штрафов, что побуждает его искать оптимальное поведение, позволяющее выполнить поставленную задачу. Этот вид МО особенно полезен в ситуациях, когда сложно или невозможно запрограммировать явные правила.

Например, в робототехнике с помощью данного вида МО можно обучать робота выполнению сложных задач, таких как захват объектов или ходьба. Кроме того, обучение с подкреплением можно применять во множестве других областей, таких как финансы, здравоохранение и транспорт, оптимизируя процессы принятия решений и повышая общую производительность. В целом обучение с подкреплением — мощный инструмент, который может произвести революцию в решении сложных задач.

Во всех этих областях машинного обучения алгоритмы служат основой и позволяют компьютерам учиться делать прогнозы и со временем улучшать качество своей работы. Поскольку МО продолжает развиваться, мы наверняка увидим еще более сложные и эффективные алгоритмы.

10.3. Алгоритмы сетевой маршрутизации

Сетевая маршрутизация — базовый телекоммуникационный процесс, в котором пакеты данных перемещаются от одного узла к другому по сложной сети до тех пор, пока не достигнут своей цели. Этот процесс имеет решающее значение, поскольку гарантирует эффективность передачи данных и доставку пакетов их получателям.

Алгоритмы играют важную роль в управлении этими сетями, оптимизируя процесс маршрутизации и гарантируя, что пакеты данных пойдут по наилучшему пути от отправителя к получателю. Эти алгоритмы разработаны с помощью сложных математических моделей и постоянно обновляются, чтобы соответствовать требованиям к передаче сетевого трафика, которые постоянно меняются.

Кроме того, протоколы сетевой маршрутизации постоянно развиваются, чтобы удовлетворить растущий спрос на более быструю передачу данных и эффективное управление сетью. Эти протоколы предназначены для управления сетевым трафиком, предотвращения заторов и обеспечения надежной и эффективной доставки пакетов данных.

Далее перечислены некоторые из наиболее часто используемых алгоритмов сетевой маршрутизации.

10.3.1. Алгоритм Дейкстры

Как вы знаете из главы 7, алгоритм Дейкстры очень эффективен в задачах поиска кратчайшего пути в сети или графе. Он часто используется в сетевой маршрутизации (в первую очередь в IP-маршрутизации) для определения наиболее эффективного маршрута передачи пакетов от одного сетевого узла к другому. Это очень полезный инструмент, если заранее известны затраты, связанные с прохождением каждого сегмента пути.

Алгоритм основан на присваивании заранее известного значения расстояния каждому узлу с последующим многократным обновлением этих значений для поиска минимума. Такой подход позволяет алгоритму выявить оптимальный маршрут для перемещения пакетов, что особенно полезно для систем, в которых требуется быстрая и эффективная маршрутизация сетевого трафика.

Помимо IP-маршрутизации, алгоритм Дейкстры используется в других областях, таких как поиск кратчайшего пути между двумя точками на карте и анализ социальных сетей для определения наиболее эффективных факторов, влияющих на продвижение продукта или услуги. В целом алгоритм Дейкстры — универсальный инструмент с широким спектром применения, и его потенциал ограничен только креативностью пользователей.

Пример

```
import networkx as nx

# Определить граф с взвешенными ребрами
G = nx.DiGraph()
G.add_edge('A', 'B', weight=1)
G.add_edge('B', 'C', weight=2)
G.add_edge('A', 'C', weight=3)

# Использовать алгоритм Дейкстры для поиска кратчайшего пути
print(nx.dijkstra_path(G, 'A', 'C')) # Выведет: ['A', 'B', 'C']
```

В этом простом примере на Python используется библиотека networkx для создания ориентированного графа с взвешенными ребрами, а затем находится кратчайший путь от `'A'` до `'C'` с помощью алгоритма Дейкстры.

10.3.2. Алгоритм Беллмана — Форда

Алгоритм Беллмана — Форда пользуется большой популярностью в сетевой маршрутизации, особенно в протоколах маршрутизации, таких как протокол маршрутной информации (Routing Information Protocol, RIP). В отличие от алгоритма Дейкстры, он прекрасно подходит для обработки графов с ребрами отрицательного веса.

Этот алгоритм производит многократную релаксацию ребер графа, обновляя значения кратчайшего пути для каждой вершины. Количество итераций, выполняемых алгоритмом, зависит от размера графа и количества ребер.

Спустя определенное количество итераций выявляются кратчайшие пути от начальной вершины к каждой второй. Алгоритм Беллмана — Форда широко применяется на практике, например для проектирования компьютерных сетей, транспортных систем и даже финансовых рынков. Способность обрабатывать ребра с отрицательными весами делает его ценным инструментом анализа сложных систем.

Пример

```
# Используется тот же граф G, что и в предыдущем примере

# Применить алгоритм Беллмана — Форда для поиска кратчайшего пути
print(nx.bellman_ford_path(G, 'A', 'C')) # Выведет: ['A', 'B', 'C']
```

Мы снова использовали библиотеку networkx, но на этот раз поиск кратчайшего пути производится с помощью алгоритма Беллмана — Форда.

10.3.3. Протокол маршрутизации по состоянию канала

Этот протокол маршрутизации особенно полезен при работе в больших и сложных сетях. В LSRP (Link State Routing Protocol — протокол маршрутизации по состоянию канала) каждый маршрутизатор поддерживает базу данных с информацией о топологии сети, в том числе обо всех каналах, узлах и их состоянии. Эта база данных постоянно обновляется, чтобы отображать текущее состояние сети.

Когда маршрутизатору необходимо переслать пакет, он применяет алгоритм Дейкстры к своей базе данных, рассматривая все возможные маршруты к месту назначения пакета. Это гарантирует, что маршрутизатор выберет наиболее надежный и эффективный путь из доступных, а не просто заранее определенный.

Имея доступ ко всей топологии сети, LSRP позволяет маршрутизаторам принимать более обоснованные решения о маршрутизации даже в динамически меняющихся средах. Это означает, что LSRP может помочь предотвратить перегрузку, уменьшить потерю пакетов и улучшить общую производительность сети.

Существует один важный принцип, которым руководствуется большинство алгоритмов маршрутизации: они стремятся найти наиболее эффективный путь передачи данных. Эффективность может оцениваться по-разному. Одни алгоритмы минимизируют расстояние, которое должны пройти данные, тогда как другие стремятся избежать перегруженных путей, уменьшить задержку или даже минимизировать стоимость передачи данных.

Помимо всего прочего, эти алгоритмы должны постоянно адаптироваться к изменениям в сети. Например, в ней могут появляться новые узлы или исчезать существующие, а объем трафика может резко меняться в течение дня. Способность быстро пересчитывать маршруты с учетом этих изменений — важная особенность надежных алгоритмов сетевой маршрутизации.

Более того, во многих сетях одновременно используется несколько алгоритмов маршрутизации. Этот подход, известный как многопутевая маршрутизация, повышает надежность и эффективность сети. Если один путь перегружен или выходит из строя, то данные можно быстро перенаправить по другому пути.

Наконец, представим сложность задачи, которую решают эти алгоритмы. Даже относительно небольшая сеть может иметь астрономическое количество потенциальных путей. Несмотря на это, алгоритмы маршрутизации могут определять эффективные пути за доли секунды. Эта скорость наглядно показывает силу алгоритмического подхода и его способности решать, казалось бы, неразрешимые задачи!

Итак, просматривая сайты в Интернете или просто отправляя электронное письмо, помните, что «за сценой» усердно работают сложные алгоритмы маршрутизации, гарантирующие, что ваши данные доберутся до места назначения максимально эффективным путем. Поэтому в следующий раз, когда ваше интернет-соединение будет работать без сбоев, найдите минутку, чтобы мысленно поблагодарить этих незаметных героев цифровой эпохи!

10.4. Практические задачи

Перейдем к решению задач, которые позволят вам закрепить новые знания, полученные в этой главе. Кроме того, вы увидите, как алгоритмы применяются в различных реальных сценариях.

Задача 1. Алгоритмы в базах данных

Вам поручено разработать систему баз данных для библиотеки. В ней хранятся тысячи книг, каждая из которых имеет уникальный идентификатор, название, список авторов и дату публикации. Вам необходимо разработать систему, которая позволит сотрудникам библиотеки:

1) добавлять в коллекцию новую книгу;

2) находить книгу по ее уникальному идентификатору;

3) составлять список всех книг, написанных конкретным автором.

Опишите структуры данных и алгоритмы, которые вы использовали бы для каждой операции, и объясните почему.

Решение

1. Добавление новой книги в коллекцию. Это простая операция вставки. Книги могут храниться в таблице базы данных, каждая запись которой соответствует одной книге. В качестве структуры данных можно использовать массив или связанный список в зависимости от деталей реализации системы базы данных.
2. Поиск книги по ее уникальному идентификатору. Для быстрого поиска можно использовать хеш-таблицу или ассоциативный массив, где роль ключа играет уникальный идентификатор, а роль значения — сведения о книге. Поиск по ключу в хеш-таблице в среднем имеет временную сложность O(1), что очень эффективно.
3. Составление списка всех книг, написанных конкретным автором. Для этой операции можно использовать структуру данных с несколькими индексами или ключами. Это позволит быстро сопоставить имя автора со списком книг за время O(1).

Задача 2. Алгоритмы искусственного интеллекта

Вы создаете чат-бот, который должен понимать смысл сообщений пользователя и уметь обрабатывать широкий спектр пользовательских запросов: от простых («Какая сейчас погода?») до более сложных («Найди ближайший ресторан мексиканской кухни, который сейчас открыт»).

Опишите, какие алгоритмы вы использовали бы для интерпретации различных видов запросов, и объясните, как они будут работать.

Решение

Алгоритмы, используемые в этом случае, будут в первую очередь алгоритмами обработки естественного языка (NLP).

Для обработки простых запросов, таких как «Какая сейчас погода?», и определения их смысла можно использовать алгоритмы сопоставления с образцом или извлечения ключевых слов.

Для обработки более сложных запросов, таких как «Найди ближайший ресторан мексиканской кухни, который сейчас открыт», можно использовать более сложные алгоритмы NLP, такие как распознавание именованных объектов (чтобы идентифицировать «ресторан мексиканской кухни» как

искомое место, «ближайший» как спецификатор местоположения и «сейчас открыт» как уточнение времени) и распознавание намерений (чтобы понять, что пользователь ищет конкретное место).

Скорее всего, эти алгоритмы будут действовать в составе некой модели, обученной на большом объеме примеров данных. Она будет принимать вводимые пользователем запросы, обрабатывать их с помощью своих алгоритмов и выводить результат интерпретации намерения.

Задача 3. Алгоритмы сетевой маршрутизации

Вы работаете сетевым инженером в компании, создающей новый центр обработки данных. Он будет подключаться к нескольким существующим сетевым узлам, и вы должны обеспечить максимально эффективную передачу данных.

Опишите, как вы будете использовать алгоритмы, чтобы:

1) определить наиболее эффективные пути передачи данных между новым центром обработки данных и существующими сетевыми узлами;
2) корректировать эти пути в режиме реального времени, оперативно реагируя на изменения условий в сети.

Решение

1. Определять наиболее эффективные пути передачи данных между новым центром обработки данных и существующими сетевыми узлами можно с помощью графовых алгоритмов, таких как алгоритм Дейкстры или алгоритм поиска A*. Они помогут отыскать кратчайший путь между узлами графа, который представляет собой сеть центров обработки данных.
2. Для оперативной корректировки путей в соответствии с изменениями условий в сети можно применять протоколы динамической маршрутизации, например OSPF (Open Shortest Path First) или BGP (Border Gateway Protocol). Они используют алгоритмы, которые могут реагировать на изменения в топологии сети и обновлять таблицы маршрутизации в режиме реального времени. Что касается конкретных алгоритмов, то BGP использует протокол вектора пути, а OSPF — алгоритм Дейкстры.

Помните, что это довольно сложные задачи, но с ними часто приходится иметь дело профессионалам в области управления базами данных, искусственного интеллекта и сетевой инженерии. Обдумывая эти задачи и определяя подходящие алгоритмы, вы сможете развить навыки, необходимые для решения подобных задач. Удачи!

Резюме

В этой главе мы исследовали алгоритмы и их влияние на различные технологии. Сначала мы увидели, насколько алгоритмы важны в базах данных, составляющих основу почти всех цифровых сервисов. Базы хранят огромные объемы информации, и алгоритмы играют ключевую роль в эффективном хранении этих данных, их извлечении и управлении ими. Мы обсудили различные структуры данных, такие как B-деревья и хеш-таблицы, а также алгоритмы индексирования, позволяющие базам работать максимально эффективно.

Затем мы переключились на тему искусственного интеллекта. ИИ преобразует различные сферы и отрасли, и в основе этого процесса лежат алгоритмы. Мы обсудили машинное обучение, в том числе алгоритмы обучения на данных без явного программирования. Подробно рассмотрели такие понятия, как деревья решений, нейронные сети и обучение с подкреплением. Мы также обсудили алгоритмы поиска, используемые в ИИ, такие как алгоритмы поиска A* и поиск в глубину (DFS), находящие оптимальные решения в предметных пространствах.

Далее мы углубились в область сетевой маршрутизации, где алгоритмы определяют наилучшие маршруты для передачи данных по сети. Мы выделили алгоритм Дейкстры, находящий кратчайший путь в сети, и алгоритм Беллмана — Форда, обслуживающий сети с отрицательными весами. Мы также узнали, что протоколы динамической маршрутизации, такие как OSPF и BGP, используют алгоритмы для адаптации к изменениям в сети и обеспечения эффективной передачи данных.

Наконец, мы закрепили полученные знания, решив несколько практических задач, каждая из которых соответствует реальным ситуациям, обсуждавшимся в этой главе. Задачи побудили нас критически осмыслить применение алгоритмов и помогли укрепить навыки решения задач.

Подытожим. Алгоритмы — невидимая сила, движущая технологическим прогрессом. Они незаменимы в цифровом мире и используются в разных областях: от организации информации в базах данных и принятия решений с помощью ИИ до определения эффективных маршрутов в сетях. Основная цель главы заключалась в том, чтобы помочь вам понять, как эти алгоритмы применяются в реальных ситуациях, и взглянуть на них с практической точки зрения.

Продолжая развиваться, помните, что разбираться в алгоритмах означает не только знать теорию или уметь писать код. Речь идет о понимании их потенциала, особенностей практического применения и в итоге о возможностях их использования для решения сложных задач и реализации инноваций. Исследования в данной области продолжаются, поэтому будьте готовы к новым знаниям и открытиям.

ЗАКЛЮЧЕНИЕ

В этой книге мы совершили путешествие по удивительному миру алгоритмов. Мы начали с простого знакомства с алгоритмами, а затем постепенно исследовали различные виды алгоритмов, методы их проектирования, структуры данных, которые они часто используют, и примеры их применения в реальных сценариях.

В начальных главах мы пошагово изучали фундаментальные понятия, анализируя внутреннюю работу различных видов алгоритмов, таких как алгоритмы поиска, сортировки и графовые алгоритмы. Мы познакомились с известными алгоритмами двоичного поиска, быстрой сортировки, сортировки слиянием, поиска в ширину, поиска в глубину, алгоритмом Дейкстры и многими другими. Мы также уделили особое внимание временной и пространственной сложностям алгоритмов, которые оказывают решающее влияние на выбор эффективных алгоритмов.

Мы исследовали различные структуры данных, используемые в алгоритмах, такие как массивы, связанные списки, стеки, очереди, деревья и графы. Важность структур данных невозможно переоценить. Они образуют фундамент, на котором строятся алгоритмы, и обеспечивают эффективное хранение и организацию данных, с которыми работают алгоритмы.

Мы познакомились с различными методами проектирования алгоритмов. Узнали о рекурсии, когда алгоритм вызывает сам себя для решения небольших экземпляров одной и той же задачи, и об итеративных подходах, использующих циклы для решения задач. Мы изучили поиск с возвратом, который подразумевает серию принятия решений и их отмену, когда выясняется, что они не приближают к цели. И наконец, обсудили метод ветвей и границ, используемый для решения задач оптимизации.

В последней части нашего путешествия мы рассмотрели вопросы применения алгоритмов в реальных сценариях, где исследовали их использование в базах данных, искусственном интеллекте и сетевой маршрутизации. Мы узнали, что алгоритмы являются движущей силой этих технологий. Они позволяют нам управлять огромными объемами данных, принимать обоснованные решения и устанавливать надежные и эффективные сетевые соединения.

В этой книге также содержится множество задач, позволяющих применить полученные знания на практике, стимулировать ваш мыслительный процесс, более глубоко понять алгоритмы и увидеть их реальную значимость.

В завершение отметим: начиная понимать алгоритмы, вы словно приобретаете мощный арсенал. Каждый алгоритм или метод, который вы изучаете, — это инструмент, который можно добавить в арсенал. Чем он обширнее, тем лучше вы подготовлены к решению множества задач. Цель этой книги — предоставить вам большой набор инструментов и показать, как их можно применять в реальных ситуациях.

Алгоритмы — не просто фрагменты кода. Это стратегии решения задач. Они составляют основу информатики, суть программирования и фундамент цифрового мира, в котором мы живем. По мере развития эпохи цифровизации важность алгоритмов будет только расти.

Мир алгоритмов огромен. Мы рассмотрели очень много тем, но это лишь верхушка айсберга. Помимо перечисленных в этой книге алгоритмов, существует много других, каждый из которых предлагает уникальный подход к решению задач. Продолжайте исследовать их и учитесь применять. Чем больше вы будете практиковаться, тем более эффективными будут ваши решения для множества задач.

Спасибо, что присоединились к нам в этом путешествии. Пусть знания и идеи, которые вы получили, послужат вам во всех ваших начинаниях. Как говорится, «путь в тысячу миль начинается с первого шага». Считайте эту книгу таким шагом. Успехов в программировании, и никогда не прекращайте учиться!

Дальнейшие действия

Если вы прочитали эту книгу и хотите получить новые знания в области программирования, то мы можем порекомендовать другие книги, изданные нашей компанией, занимающейся разработкой ПО. Они охватывают широкий круг тем и помогут вам совершенствовать навыки программирования.

1. *Guest C., Chandra B. K. S., Shaw B., Badhwar S., Bird A.* Master Web Development with Django (Packt, 2021). Это подробное руководство по созданию веб-приложений с использованием Django, одной из самых популярных веб-платформ на Python. Книга описывает все нюансы: от настройки среды разработки до развертывания приложения на рабочем сервере.

2. *Horton A., Vice R.* Mastering React (Packt, 2016). React — популярная библиотека JavaScript для создания пользовательских интерфейсов. Эта книга поможет вам освоить основные идеи React и покажет, как создавать мощные и динамичные веб-приложения.
3. *Taieb D.* Data Analysis with Python: A Modern Approach (Packt, 2018). Python — язык программирования, прекрасно подходящий для анализа данных, и эта книга поможет вам полностью раскрыть его потенциал. В ней представлены такие темы, как первичная обработка данных, манипулирование данными и их визуализация, а также содержатся упражнения, которые помогут вам попрактиковаться в применении полученных знаний.
4. *Zollanvari A.* Machine Learning with Python (Springer, 2023). Машинное обучение — одна из самых интересных областей информатики, и эта книга поможет вам начать создавать собственные модели машинного обучения с использованием Python. В ней описываются такие темы, как линейная регрессия, логистическая регрессия и деревья решений.
5. *Cuantum Technologies.* Mastering ChatGPT and Prompt Engineering (2023). В этой книге вы совершите увлекательное путешествие по миру инженерии подсказок, начиная с основ языковых моделей искусственного интеллекта и заканчивая сложными стратегиями и реальными приложениями.

Благодаря этим книгам вы сможете и дальше развивать навыки программирования и изучать язык Python. Мы считаем, что программирование — это навык, который можно освоить и совершенствовать с течением времени, и стараемся предоставлять ресурсы, с помощью которых вы сможете достичь своих целей.

Мы также хотели бы воспользоваться этой возможностью и поблагодарить вас за то, что выбрали нашу компанию в качестве проводника по миру программирования. Надеемся, что эта книга для начинающих окажется для вас полезной, и с воодушевлением ждем возможности предоставить вам другие высококачественные ресурсы по программированию. Если у вас есть отзывы или предложения по грядущим книгам или ресурсам, пожалуйста, не стесняйтесь и обращайтесь к нам (https://www.cuantum.tech/contact/). Мы будем рады услышать вас!

Cuantum Technologies

Алгоритмы с нуля

Перевел с английского А. Киселев

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Т. Никифорова, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 06.03.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000 экз. Заказ Х-610.
Отпечатано в типографии ООО «Экопейпер», 420044, Россия, г. Казань, пр. Ямашева, д. 36Б.

Алгоритмы

с нуля

Погрузитесь в мир алгоритмов! Разберитесь в их принципах, особенностях проектирования и практического применения.

Вы познакомитесь с различными видами алгоритмов, узнаете их сильные и слабые стороны и поймете, в каких контекстах они лучше всего работают. На практических примерах увидите, как эти мощные инструменты используются для решения задач в информатике, анализе данных, искусственном интеллекте и других областях.

Каждая глава содержит понятные объяснения, наглядные примеры и задачи, помогающие закрепить изученный материал. Особый акцент сделан на вычислительном мышлении и анализе эффективности алгоритмов — важнейших навыках в области современных технологий.

Книга «Алгоритмы с нуля» послужит ценным ресурсом и для новичков, и для профессионалов, желающих отточить свои навыки.