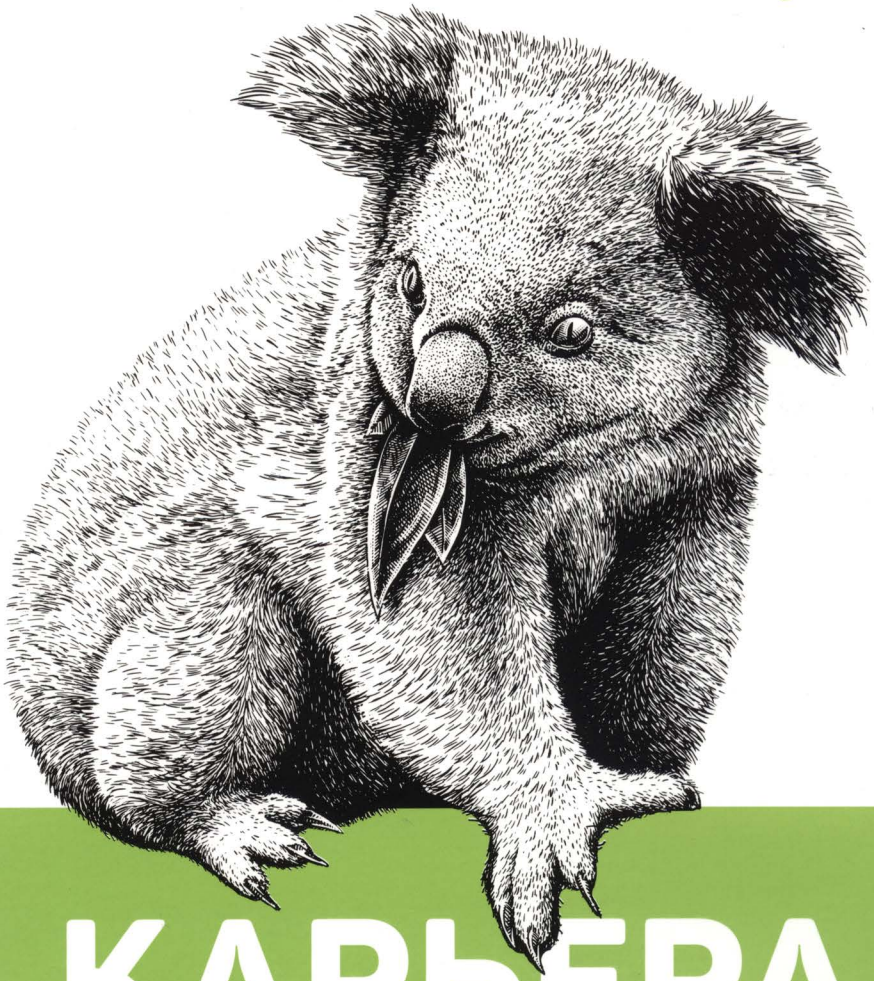
БИБЛИОТЕКА ПРОГРАММИСТА

6-е издание

Г. Лакман Макдауэлл



КАРЬЕРА ПРОГРАММИСТА

БЕСТСЕЛЛЕР AMAZON

#1 в разделе ПРОХОЖДЕНИЕ СОБЕСЕДОВАНИЯ
#1 в разделе РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
#1 в разделе СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

СКАЧАНО С WWW.SHAREWOOD.BIZ - ПРИСОЕДИНЯЙСЯ!

БИБЛИОТЕКА
ПРОГРАММИСТА

CRACKING
the
CODING INTERVIEW

6th Edition

189 Programming Questions and Solutions

GAYLE LAAKMANN MCDOWELL
Founder and CEO, CareerCup.com

CareerCup, LLC
Palo Alto, CA

БИБЛИОТЕКА ПРОГРАММИСТА

Г. Лакман Макдауэлл

КАРЬЕРА ПРОГРАММИСТА

ПРОХОЖДЕНИЕ СОБЕСЕДОВАНИЯ
РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

ПИТЕР
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2020

Гэйл Лакман Макдауэлл

Карьера программиста

6-е издание

Серия «Библиотека программиста»

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Корректоры	*С. Беляева, Н. Викторова, М. Молчанова*
Художник	*С. Заматевская*
Верстка	*Л. Соловьева*

ББК 32.973.2
УДК 004.3

Лакман Макдауэлл Г.

Л19 Карьера программиста. 6-е изд. — СПб.: Питер, 2020. — 688 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-03239-1

Книга «Карьера программиста» основана на опыте практического участия автора во множестве собеседований, проводимых лучшими компаниями. Это квинтэссенция сотен интервью со множеством кандидатов, результат ответов на тысячи вопросов, задаваемых кандидатами и интервьюерами в ведущих мировых корпорациях. Из тысяч возможных задач и вопросов в книгу были отобраны 189 наиболее интересных и значимых. Шестое издание этого мирового бестселлера поможет вам наилучшим образом подготовиться к собеседованию при приеме на работу программистом или руководителем в крупную IT-организацию или перспективный стартап. Основную часть книги составляют ответы на технические вопросы и задания, которые обычно получают соискатели на собеседовании в таких компаниях, как Google, Microsoft, Apple, Amazon и других. Рассмотрены типичные ошибки, которые допускают кандидаты, а также эффективные методики подготовки к собеседованию. Используя материал этой книги, вы с легкостью подготовитесь к устройству на работу в Google, Microsoft или любую другую ведущую IT-компанию.

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

ISBN 978-0984782857 англ.
ISBN 978-5-496-03239-1

© 2015 by CareerCup
© Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Библиотека программиста», 2020

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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: 10.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — 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 20.09.19. Формат 70×100/16. Бумага писчая. Усл. п. л. 55,470. Доп. тираж. Заказ 7640

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru
тел: 8(499) 270-73-59

Оглавление

Предисловие **11**

Введение **12**

Что-то пошло не так 12

Мой подход 13

Моя страсть 13

Часть I. Процесс собеседования **14**

Почему? 15

Как выбираются вопросы 17

Часто задаваемые вопросы 18

Часть II. За кулисами **19**

Microsoft 20

Amazon 21

Google 22

Apple 23

Facebook 24

Palantir 25

Часть III. Нестандартные случаи **27**

Профессионал 27

Тестеры и SDET 27

Менеджеры программ и менеджеры продукта 28

Ведущие разработчики и менеджеры 30

Стартапы 31

Для интервьюеров 32

Часть IV. Перед собеседованием **37**

Получаем «правильный» опыт 37

Идеальное резюме 38

Часть V. Подготовка к поведенческим вопросам **41**

Поведенческие вопросы 41

Ответы на поведенческие вопросы 43

Часть VI. «O» большое 47
Аналогия 47
Временная сложность 47
Пространственная сложность 49
Константы 50
Исключение второстепенных факторов 51
Составные алгоритмы: сложение и умножение 52
Амортизированное время 52
Сложность Log N 53
Сложность рекурсивных алгоритмов 54

Часть VII. Технические вопросы 56
Как организовать подготовку 56
Что нужно знать 56
Процесс решения задачи 58
Метод оптимизации 1: поиск BUD 63
Метод оптимизации 2: интуитивный подход 66
Метод оптимизации 3: упрощение и обобщение 66
Метод оптимизации 4: базовый случай и расширение 67
Метод оптимизации 5: мозговой штурм структур данных 67
Неправильные ответы 68
Если вы уже знаете ответ 69
«Идеальный» язык для собеседований 69
Как выглядит хороший код 70
Не сдавайтесь! 71

Часть VIII. После собеседования 72
Реакция на предложение и на отказ 72
Предложение работы 73
Переговоры 75
На работе 76

Часть IX. Вопросы собеседования 78

1. Массивы и строки 79
Хеш-таблицы 79
ArrayList и динамические массивы 80
StringBuilder 81
Вопросы собеседования 82

2. Связные списки ... 84
Создание связного списка ... 84
Удаление узла из односвязного списка ... 85
Метод бегунка ... 85
Рекурсия и связные списки ... 86
Вопросы собеседования ... 86

3. Стеки и очереди ... 88
Реализация стека ... 88
Реализация очереди ... 89
Вопросы собеседования ... 90

4. Деревья и графы ... 92
Разновидности деревьев ... 92
Бинарные деревья и бинарные деревья поиска ... 93
Обход бинарного дерева ... 95
Бинарные кучи (min-кучи и max-кучи) ... 96
Нагруженные (префиксные) деревья ... 97
Графы ... 98
Список смежности ... 98
Поиск в графе ... 100
Вопросы интервью ... 102

5. Операции с битами ... 105
Расчеты на бумаге ... 105
Биты: трюки и факты ... 106
Поразрядное дополнение и отрицательные числа ... 106
Арифметический и логический сдвиг ... 106
Основные операции: получение и установка бита ... 107
Вопросы собеседования ... 109

6. Головоломки ... 111
Простые числа ... 111
Вероятность ... 113
Начинайте говорить ... 115
Правила и шаблоны ... 115
Балансировка худшего случая ... 116
Алгоритмический подход ... 117
Вопросы собеседования ... 117

7. Объектно-ориентированное проектирование 120
Как подходить к решению заданий 120
Паттерны проектирования 121
Вопросы собеседования 122

8. Рекурсия и динамическое программирование 125
С чего начать 125
Решения рекурсивные и итеративные 126
Динамическое программирование и мемоизация 126
Вопросы собеседования 130

9. Масштабируемость и проектирование систем 133
Работа с вопросами 133
Проектирование: шаг за шагом 134
Масштабируемые алгоритмы: шаг за шагом 136
Ключевые концепции 137
Дополнительные факторы 140
Идеальных систем не бывает 140
Пример: найдите все документы, содержащие список слов 141
Вопросы собеседования 143

10. Сортировка и поиск 145
Распространенные алгоритмы сортировки 145
Алгоритмы поиска 148
Вопросы собеседования 149

11. Тестирование 152
Чего ожидает интервьюер 152
Тестирование реального объекта 153
Тестирование программного обеспечения 154
Тестирование функций 156
Поиск и устранение неисправностей 157
Вопросы собеседования 158

12. C и C++ 159
Классы и наследование 159
Конструкторы и деструкторы 160
Виртуальные функции 160
Виртуальный деструктор 161
Значения по умолчанию 162
Перегрузка операторов 163

Указатели и ссылки ... 163
Шаблоны ... 164
Вопросы собеседования ... 165

13. Java ... 167
Подход к изучению ... 167
Перегрузка vs переопределение ... 167
Java Collection Framework ... 168
Вопросы собеседования ... 169

14. Базы данных ... 171
Синтаксис SQL и его варианты ... 171
Денормализованные и нормализованные базы данных ... 171
Команды SQL ... 172
Проектирование небольшой базы данных ... 174
Проектирование больших баз данных ... 175
Вопросы собеседования ... 175

15. Потоки и блокировки ... 177
Потоки в Java ... 177
Синхронизация и блокировки ... 179
Взаимные блокировки и их предотвращение ... 182
Вопросы собеседования ... 183

16. Задачи умеренной сложности ... 185

17. Сложные задачи ... 190

Часть X. Решения ... 196

1. Массивы и строки ... 197

2. Связные списки ... 214

3. Стеки и очереди ... 234

4. Деревья и графы ... 248

5. Операции с битами ... 286

6. Головоломки ... 299

7. Объектно-ориентированное проектирование ... 317

8. Рекурсия и динамическое программирование ... 355

9. Масштабируемость и проектирование систем ... 387

10. Сортировка и поиск 415

11. Тестирование 437

12. C и C++ 443

13. Java 456

14. Базы данных 465

15. Потоки и блокировки 472

16. Задачи умеренной сложности 488

17. Сложные задачи 560

Часть XI. Дополнительные материалы 665

Полезные формулы 666
Топологическая сортировка 668
Алгоритм Дейкстры 669
Разрешение коллизий в хеш-таблице 672
Поиск подстроки по алгоритму Рабина — Карпа 673
АВЛ-деревья 674
Красно-черные деревья 676
MapReduce 680
Дополнительные материалы 681

Часть XII. Библиотека кода 683

HashMapList<T, E> 683
TreeNode (бинарное дерево поиска) 684
LinkedListNode (связный список) 685
Trie и TrieNode 686

Часть XIII. Подсказки
(скачайте с сайта издательства www.piter.com) **689**

1. Структуры данных 690

2. Концепции и алгоритмы 699

3. Вопросы, основанные на знаниях 713

4. Дополнительные задачи 716

http://goo.gl/ssQdRk

Предисловие

Дорогой читатель!

Я не HR-менеджер и не работодатель, а всего лишь разработчик программного обеспечения. Именно поэтому я знаю, что может произойти на собеседовании (например, вас попросят быстренько разработать блестящий алгоритм, а затем написать к нему безупречный код). Мне самой давали такие же задания, когда я проходила собеседование в Google, Microsoft, Apple, Amazon и в других компаниях.

Случалось мне быть и по другую сторону баррикад — я проводила собеседования, просматривала стопки резюме соискателей, занимаясь подбором персонала. Я оценивала то, как они решали — или пытались решать — сложные задачи. Я спорила в комитете по набору персонала Google, достаточно ли хорошо показал себя кандидат для получения работы. Именно поэтому я с полной уверенностью могу утверждать, что мне знакомы все тонкости процесса найма, потому что я неоднократно прошла его полностью.

Если вы читаете эту книгу, то это означает, что вы собираетесь пройти собеседование завтра, на следующей неделе или через год. И я постараюсь укрепить ваше понимание основ компьютерной теории, а затем покажу, как применять их для успешного прохождения собеседований.

В 6-м издании книги материал 5-го издания был дополнен более чем на 70%: дополнительные вопросы, обновленные решения, введения к главам, новые стратегии алгоритмов, подсказки ко всем задачам и другие материалы. Обязательно загляните на сайт CrackingTheCodingInterview.com, там вы можете пообщаться с другими соискателями и получить новую информацию.

Навыки, которые мы будем развивать, принесут огромную пользу. Хорошая подготовка позволит вам расширить ваши технические и коммуникативные способности, а это никогда не бывает лишним.

Внимательно прочитайте вводные главы. Возможно, именно приведенный в них материал сыграет ключевую роль в принятии решения о вашем найме на работу.

И помните: собеседование будет сложным! В свое время (в период моей работы в Google) я видела многих интервьюеров, одни из них задавали «легкие» вопросы, а другие — «сложные». И знаете что? Простые вопросы вовсе не означали, что кандидату будет проще получить работу. Главное — не безупречные ответы на вопросы (такое бывает очень редко!). Главное, чтобы ваш ответ был *лучше, чем у других* кандидатов. И не паникуйте, если вам достался сложный вопрос, — те, кто его задают, знают, что вопрос сложен и не ждут от вас идеального ответа.

Читайте, учитесь, а я желаю вам удачи!

Гэйл Лакман Макдауэлл,
основатель CareerCup.com

Введение

Что-то пошло не так

Очередное собеседование обернулось разочарованием... в очередной раз. Никто из десяти кандидатов не получил работу. Может быть, мы были слишком строги?

Я была особенно огорчена: мы отказали одному из *моих* кандидатов. Мой бывший студент. Тот, кого я рекомендовала. У него был достаточно высокий средний балл в Вашингтонском университете — одной из лучших школ мира по компьютерным дисциплинам, — и он активно занимался проектами с открытым кодом. Он был энергичен, сообразителен, обладал творческим мышлением, упорно трудился и был компьютерным фанатом в хорошем смысле этого слова.

Но я была вынуждена согласиться с мнением других членов комиссии: он показал себя не лучшим образом. Даже если бы сыграла свою роль моя рекомендация, моему ученику все равно отказали бы на более поздних этапах отбора. Слишком много было «красных» карточек.

Несмотря на свой ум, кандидат с трудом справлялся с поставленными задачами. Более успешные кандидаты быстро разобрались с первым вопросом, который был построен на известной задаче, а у моего студента возникли проблемы с разработкой алгоритма. Когда он наконец-то осилил алгоритм, то не учел возможность оптимизации для других сценариев. Когда дело дошло до написания кода, он допустил множество ошибок. Это был не худший кандидат, но все видели, как ему далеко до победного результата.

Через пару недель он позвонил мне, а я не знала, что сказать. Нужно стать еще умнее? Дело было не в этом, я знала, что у него блестящий ум. Научиться лучше программировать? Нет, его навыки были не хуже, чем у других программистов, которых я знала.

Он тщательно готовился, как и большинство кандидатов. Он изучил классический учебник Кернигана и Ричи, он прочитал CLRS[1]. Он может описать в подробностях множество способов балансировки дерева и умеет делать на C такое, на что не осмелится ни один нормальный программист.

Мне пришлось сказать ему горькую правду — книжного академического образования недостаточно. Книги — это замечательно, но они не помогут вам пройти собеседование. Почему? Подскажу: интервьюеры не видели красно-черных деревьев со времен *своего* обучения в университете.

Чтобы успешно пройти собеседование, нужно готовиться на *реальных* вопросах, встречающихся на собеседованиях. Нужно решать *реальные* задачи и изучать

[1] CLRS (Thomas H. **C**ormen, Charles E. **L**eiserson, Ronald L. **R**ivest, Clifford **S**tein) — Introduction to Algorithms (*Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.* Алгоритмы. Построение и анализ) — популярнейшая книга по алгоритмам и структурам данных.

встречающиеся закономерности. Главное — разработка новых алгоритмов, а не запоминание существующих задач.

Книга «Карьера программиста» основана на опыте моего практического участия во множестве собеседований, проводимых лучшими компаниями. Это квинтэссенция сотен интервью с множеством кандидатов, результат ответов на тысячи вопросов, задаваемых кандидатами и интервьюерами в ведущих мировых корпорациях. В эту книгу из тысяч возможных задач и вопросов были отобраны 189 наиболее интересных.

Мой подход

В данной книге основное внимание уделено задачам алгоритмизации, программирования и дизайна. Почему? Потому что ответы на «поведенческие» вопросы могут быть такими же разнообразными, как и ваше резюме. И хотя в некоторых фирмах задают вопросы на эрудицию (например, «Что такое виртуальная функция?»), навыки, полученные в ходе подготовки к таким вопросам, ограничены весьма узкими областями. Я расскажу и о таких вопросах, но прежде всего я хотела бы уделить внимание более сложным вещам.

Моя страсть

Преподавание — моя страсть. Мне нравится помогать людям совершенствоваться и узнавать новое.

Свой первый «официальный» преподавательский опыт я получила в колледже Пенсильванского университета на должности ассистента преподавателя, это был курс информатики. Как техническому специалисту Google, мне всегда нравилось обучать и курировать новые кадры. Я даже использовала свои 20 % времени для преподавания двух новых курсов информатики в Вашингтонском университете.

Прошли годы. Теперь я снова занимаюсь преподаванием информатики, но на этот раз с более конкретной целью — для подготовки технических специалистов к приему на работу. Я видела их ошибки и те проблемы, с которыми они сталкиваются, и разработала методы и стратегии для их преодоления.

В моих книгах отражена моя страсть к преподаванию. Даже сейчас меня можно время от времени найти на CareerCup.com, где я помогаю пользователям, обратившимся за содействием.

Присоединяйтесь к нам!

Гэйл Лакман Макдауэлл

От издательства

Чтобы скачать дополнительные материалы к книге, воспользуйтесь QR-кодом или короткой ссылкой. 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.

http://goo.gl/ssQdRk

I

Процесс собеседования

В большинстве ведущих технических (и многих других) компаний основную часть процесса собеседования составляют вопросы по алгоритмам и программированию. Считайте, что это проверка навыков практического решения задач. Интервьюер старается оценить вашу способность решать алгоритмические задачи, которые еще не встречались вам ранее.

Очень часто все собеседование занимает только один вопрос. Сорок пять минут — не так уж много. Трудно решить за это время несколько разных задач.

В процессе решения постарайтесь рассуждать вслух и объяснить свой ход мыслей. Иногда интервьюер может вступить в разговор, чтобы помочь вам; не сопротивляйтесь. Это нормально; не думайте, что у вас что-то не получается. (Хотя конечно, если подсказки вам не нужны, тем лучше.)

В конце собеседования интервьюер отойдет с неким внутренним ощущением того, как вы справились. Вашему результату может быть присвоена числовая оценка, но не стоит относиться к ней как к количественной метрике. Не существует таблицы, в которой было бы указано, сколько баллов должно начисляться за разные достижения. Система работает не так. Вместо этого интервьюер оценивает результат кандидата, обычно с учетом следующих факторов:

Аналитические навыки: понадобилась ли вам помощь при решении задачи? Насколько эффективным было решение? Сколько времени понадобилось для того, чтобы прийти к решению? Если вам пришлось проектировать/разрабатывать архитектуру нового решения, насколько хорошо вы структурировали задачу и продумали последствия выбираемых вариантов?

Навыки программирования: удалось ли вам успешно преобразовать алгоритм в код приемлемого качества? Был ли этот код чистым и хорошо структурированным? Были ли учтены потенциальные ошибки? Насколько хорошим был стиль программирования?

Технические знания/основы компьютерных технологий: обладаете ли вы хорошими базовыми знаниями в области компьютеров и других сопутствующих технологий?

Опыт: принимали ли вы удачные технические решения в прошлом? Строили ли интересные, сложные проекты? Проявляли инициативу, энтузиазм и другие важные качества?

Соответствие корпоративной культуре/коммуникационные навыки: насколько хорошо ваша личность и ваши ценности соответствуют духу компании и команды? Удалось ли вам установить контакт с интервьюером?

Важность этих аспектов сильно зависит от вопроса, интервьюера, роли, команды и компании. В стандартных вопросах по алгоритмам она практически полностью приходится на первые три вопроса.

Почему?

Это один из самых частых вопросов, возникающих у кандидатов в начале процесса. Почему все происходит именно так? Ведь если задуматься:

1. Многие замечательные кандидаты показывают не лучшие результаты на подобных собеседованиях.
2. Ответ на вопрос может быть известен заранее.
3. Такие структуры данных, как бинарные деревья поиска, относительно редко используются в практическом программировании, а если потребуется, их можно изучить в процессе.
4. Программирование «за доской» неестественно. В жизни код никогда не пишется фломастером на доске.

Все эти претензии не лишены смысла. Более того, я соглашусь с ними по всем пунктам — по крайней мере частично.

В то же время такой подход оправдан для некоторых — хотя и не для всех — должностей. Неважно, согласны вы с этой логикой или нет, но желательно понять, почему задаются такие вопросы, а для этого стоит немного разобраться в менталитете интервьюера.

Ложно-отрицательные решения приемлемы

Как ни печально (особенно для кандидатов), это правда.

С точки зрения компании вполне приемлемо, что некоторые хорошие кандидаты будут отклонены. Компания стремится создать хорошее сообщество работников. Она допускает, что некоторые хорошие люди будут упущены. Конечно, ложных отрицательных решений хотелось бы избежать, так как это повышает затраты на подбор кадров, и все же такой компромисс допустим — при условии, что в итоге удастся нанять достаточно хороших людей.

Куда большую угрозу создают ложные положительные решения: люди, которые хорошо показывают себя на собеседовании, а на самом деле плохо подходят для работы.

Навыки практического решения задач важны

Если вы можете решить несколько сложных задач (возможно, с некоторой помощью), то, скорее всего, вы неплохо справитесь с разработкой оптимальных алгоритмов. Проще говоря, вы умны.

Умные люди создают хорошие продукты, а это важно для компании. Конечно, это не единственный важный фактор, но и не последний.

Знание основных структур данных и алгоритмов полезно

Многие интервьюеры считают, что знание основ компьютерной теории действительно полезно. Деревья, графы, списки, сортировка — все это периодически встречается в реальной работе. И в таких ситуациях знания действительно очень важны.

Можно ли изучить материал при необходимости? Бесспорно. Однако очень трудно понять, что в данной ситуации может пригодиться бинарное дерево поиска, если вы не знаете о его существовании, а если вы знаете о его существовании, то, скорее всего, понимаете и основные механизмы его работы.

Другие интервьюеры объясняют внимание к структурам данных и алгоритмам тем, что это хороший «косвенный показатель». Хотя эти знания не так трудно приобрести самостоятельно, считается, что они присущи многим хорошим разработчикам. Это означает, что разработчик либо прошел специальный курс обучения (а в этом случае приобрел достаточно широкие технические знания), либо изучал эту тему самостоятельно. В любом случае это хороший признак.

Есть еще одна причина, по которой тема структур данных и алгоритмов поднимается на собеседованиях: трудно найти вопрос, связанный с решением задач, в котором бы *не встречались* структуры данных и алгоритмы. Эти базовые сведения встречаются в абсолютном большинстве практических задач. Когда информация известна большинству кандидатов, у интервьюеров входит в привычку задавать вопросы по этой теме.

Доска помогает сосредоточиться на главном

Написать идеальный код на доске очень трудно — и это правда. К счастью, интервьюер от вас этого не ждет. Практически у всех кандидатов встречаются ошибки или незначительные синтаксические недостатки.

Доска хороша тем, что она в некоторых отношениях помогает выделить «общую картину» происходящего. Компилятора нет, поэтому добиваться того, чтобы код откомпилировался, не нужно. Не нужно писать определение класса и шаблонный код. Кандидат должен сосредоточиться на интересных, содержательных частях кода, то есть на сути вопроса.

Это не значит, что вы можете ограничиваться псевдокодом или правильность кода вообще не играет роли. Большинство интервьюеров не соглашается на псевдокод, и чем меньше ошибок, тем лучше.

Кроме того, у доски кандидату обычно приходится больше говорить и пояснять ход своих мыслей. Когда кандидат сидит за компьютером, интенсивность общения существенно снижается.

Все эти рекомендации не универсальны

Все сказанное поможет вам понять ход мышления интервьюера.

Что я об этом думаю? В правильной ситуации и при правильном исполнении такой подход к собеседованию позволит адекватно оценить навыки решения задач, а те люди, которые хорошо проходят собеседование, обычно оказываются достаточно умными.

Впрочем, собеседования нередко проходят не лучшим образом. Вам может попасться плохой интервьюер, который не умеет задавать вопросы.

Кроме того, процесс уместен не для всех компаний. В некоторых компаниях на первое место ставится предшествующий опыт или навыки владения конкретными технологиями. Вопросы такого рода не уделяют особого внимания таким аспектам.

Также традиционная форма собеседования не оценивает трудовую этику или умение сосредоточиться. Впрочем, ни один процесс собеседования не сможет полноценно проверить эти стороны кандидата.

Процесс ни в коем случае не идеален, но есть ли вообще хоть что-нибудь идеальное? У всех процессов есть свои недостатки. Короче говоря, процесс собеседования таков, каков есть, и мы должны справиться с ним как можно лучше.

Как выбираются вопросы

Кандидаты часто спрашивают меня, какие вопросы задавались на последнем собеседовании в той или иной компании, наивно полагая, что вопросы не меняются.

В абсолютном большинстве компаний нет списков вопросов, которые должны задаваться на собеседованиях. Каждый интервьюер выбирает вопросы самостоятельно.

Из-за полной свободы выбора вопросов нет ничего, из-за чего вопрос можно было бы назвать «вопросом на последних собеседованиях в Google». Просто какой-то интервьюер, работающий в Google, задал этот вопрос на одном из последних собеседований.

Вопросы, задаваемые в Google в этом году, мало чем отличаются от вопросов, задававшихся три года назад. Собственно, вопросы, задаваемые в Google, мало чем отличаются от вопросов в аналогичных компаниях — Amazon, Facebook и т. д.

Конечно, между компаниями существуют серьезные различия. Некоторые компании уделяют особое внимание алгоритмам (часто с примесью системного проектирования), а другие предпочитают вопросы, основанные на знаниях. Однако в рамках каждой конкретной категории вопросов нет ничего, что бы позволяло связать вопрос с одной компанией, а не с другой. Вопрос по алгоритмам в Google ничем принципиально не отличается от вопроса по алгоритмам в Facebook.

Все относительно

Если оценочной шкалы не существует, как оцениваются кандидаты? Как интервьюер знает, чего ожидать от вас?

Хороший вопрос. И ответ на него выглядит вполне логично, как только вы его поймете: интервьюеры оценивают вас относительно других кандидатов с тем же вопросом у того же интервьюера.

Предположим, вы придумали классную новую задачу или головоломку. Вы задаете вопрос своему другу Алексу, и он приходит к ответу через 30 минут. Белле для ответа на тот же вопрос требуется 50 минут. Крису вообще не удается решить задачу. Декстеру хватает всего 15 минут, но вы даете ему несколько серьезных подсказок; без них он бы вряд ли далеко ушел. Элли решает задачу за 10 минут и притом

предлагает альтернативное решение, которое вам даже не пришло в голову. Наконец, Фреду потребовалось 35 минут.

Вы делаете вывод: «Да, Элли справилась действительно хорошо. Вероятно, она очень хорошо разбирается в математике». (Хотя, возможно, ей просто повезло. а может быть, не повезло Крису. Можно задать еще несколько вопросов — просто чтобы убедиться, что успех не связан с везением.)

С вопросами интервью все обстоит примерно так же. Интервьюер оценивает ваши результаты, сравнивая их с результатами других людей. И речь идет не о кандидатах, с которыми он проводил собеседование на этой неделе. Речь идет о кандидатах, которым когда-либо задавался этот вопрос.

Это означает, что сложные вопросы не всегда плохи. Если вопрос сложен для вас, то он сложен и для всех. Вероятность успешного прохождения собеседования от этого не снижается.

Часто задаваемые вопросы

После собеседования мне ничего не сказали. Это отказ?

Нет. Задержка с ответом может объясняться разными причинами. Простейшее объяснение — один из интервьюеров еще не предоставил обратную связь. Лишь очень немногие компании не связываются с кандидатами, которым отказано в приеме на работу.

Если вы не получили ответа от компании в течение 3–5 дней после собеседования, (вежливо) обратитесь за информацией к специалисту по подбору кадров.

Могу ли я попытаться еще раз, если мне отказали?

Практически всегда можно сделать еще одну попытку, но нужно сделать паузу (обычно от полугода до года). Обычно плохие результаты не оказывают особого влияния на результаты повторного собеседования. Многие люди, получившие отказ от Google или Microsoft, позднее были приняты на работу в эти компании.

II

За кулисами

Многие компании проводят собеседования по очень похожим схемам. В этой главе приводится обзор того, как строятся собеседования в разных компаниях и на что обращают внимание интервьюеры. Эта информация должна определять ход вашей подготовки к собеседованию, а также ваши реакции во время собеседования и после него.

После того как вас выберут для собеседования, обычно устраивается предварительное собеседование (screening interview). Чаще всего его проводят по телефону. Кандидаты-студенты могут проходить его при личной встрече.

Пусть название вас не обманывает; на «предварительном» собеседовании часто задаются вопросы по программированию и алгоритмам, и требования могут быть такими же высокими, как при личном собеседовании. Если вы не уверены в том, будет ли собеседование направлено на техническую сторону, спросите координатора, какую должность занимает интервьюер (или какие темы будут рассматриваться в ходе собеседования). Как правило, технический специалист проводит техническое собеседование.

Многие компании используют синхронизированные редакторы документов, но другие ожидают, что вы запишете код на бумаге и прочитаете его по телефону. Некоторые даже выдают «домашнее задание», которое выполняется после завершения разговора, или просто предлагают переслать написанный вами код по электронной почте.

Как правило, кандидат проходит одно или два предварительных собеседования, прежде чем его приглашают в компанию.

В ходе собеседований в компании обычно проводятся от 3 до 6 личных собеседований, причем одно из них часто проходит за обедом. «Обеденное» собеседование обычно не имеет технической природы, а интервьюер может даже не предоставлять отчет. Как правило, на таких собеседованиях можно обсудить ваши интересы и побольше узнать о культуре данной компании. Другие собеседования в основном ориентированы на техническую сторону, и в них встречаются вопросы по программированию, алгоритмам, проектированию/архитектуре, а также поведенческие вопросы и вопросы на проверку практического опыта.

Распределение вопросов между этими темами зависит от компаний и даже рабочих групп. На него влияют приоритеты компании, ее размер, да и просто случайные факторы. Интервьюерам часто предоставляется значительная свобода действий при выборе вопросов.

После собеседования интервьюеры предоставляют отчет в той или иной форме. В некоторых компаниях интервьюеры встречаются, обсуждают ваши результаты и выносят решение. В других компаниях интервьюер передает рекомендацию

специалисту или комитету по подбору персонала для принятия окончательного решения. В третьих компаниях интервьюер решений вообще не принимает, а его мнение передается комитету по подбору персонала.

Большинство компаний связывается с кандидатом примерно через неделю для проведения следующих действий (предложения работы, отказа, дальнейшего собеседования или просто информирования о ходе процесса). Некоторые реагируют намного быстрее (иногда в тот же день!), у других это занимает больше времени.

Если вы ждали больше недели, попробуйте связаться со своим специалистом по подбору кадров. Если он не отвечает, это не значит, что вам отказано (по крайней мере во всех крупных технических компаниях и почти во всех остальных). Еще раз повторю: отсутствие ответа ничего не говорит о принятом решении. Предполагается, что кандидаты будут оповещены при принятии окончательного решения.

Если происходит задержка, обратитесь к специалисту по подбору кадров, но будьте вежливы и корректны. Эти люди ничем не отличаются от вас: они тоже бывают заняты и забывчивы.

Microsoft

Microsoft нужны умные люди — фанаты, увлеченные технологиями. Скорее всего, здесь не потребуют от вас знания всех тонкостей C++ API, но ожидают, что вы в состоянии написать код.

Итак, типичное собеседование от Microsoft. Однажды утром вы появляетесь в офисе и заполняете предварительные документы. Затем вас ждет короткое собеседование со специалистом по подбору кадров, который задаст ряд несложных вопросов. Задача этого специалиста — подготовить вас к интервью, а не мучить техническими вопросами. Он должен помочь вам собраться, для того чтобы вы меньше нервничали на настоящем собеседовании.

Будьте вежливы со специалистом по подбору кадров. Он будет вашим помощником и может даже устроить повторное собеседование в том случае, если вы провалитесь на первом. Он может отстаивать ваши интересы и замолвить слово в вашу пользу — или наоборот!

Вам предстоят четыре или пять собеседований, зачастую с разными командами интервьюеров. В отличие от других компаний, где вы встречаетесь с экзаменаторами в конференц-зале, вы будете беседовать с интервьюерами в их офисе. Рассматривайте эти собеседования как возможность осмотреться и прочувствовать корпоративную культуру.

В зависимости от конкретной ситуации интервьюеры могут поделиться информацией о том, какое впечатление вы на них произвели (или, наоборот, не сказать).

Когда интервью будет закончено, с вами может побеседовать специалист из отдела кадров, но только если вы успешно прошли собеседование. Знайте, если вы увидели менеджера по кадрам — это хороший знак!

Решение вам сообщат в течение дня или, максимум, недели. Если неделя прошла, а вы не получили никаких известий, напомните о себе по электронной почте.

Не забывайте, что отсутствие ответа может означать лишь то, что ваш специалист по подбору кадров очень занятой человек, а не то, что вам отказали.

ПОДГОТОВИТЬСЯ! *ПОЧЕМУ ВЫ ХОТИТЕ РАБОТАТЬ ИМЕННО В MICROSOFT?*

В ответе на этот вопрос Microsoft хочет услышать, что вы увлечены их технологиями. Лучший ответ выглядит примерно так: «Я всегда использовал программное обеспечение Microsoft и действительно впечатлен тем, как Microsoft удается создавать превосходные программные продукты. Например, я использовал Visual Studio для программирования игр, а API просто превосходны». Покажите свою страсть к технологии!

ОСОБЕННОСТИ С вами побеседует менеджер по кадрам только в том случае, если вы прошли собеседование. Расценивайте разговор с менеджером как хороший знак.

Кроме того, Microsoft предоставляет командам больше самостоятельности, а набор продуктов весьма разнообразен. Конкретные ситуации могут сильно различаться, потому что разные команды руководствуются разными критериями.

Amazon

В Amazon процесс обычно начинается с пары предварительных интервью по телефону, во время которых кандидат беседует с разными командами. Впрочем, собеседований может быть и больше — это означает, что интервьюеры сомневаются или же вас рассматривают как кандидата для нескольких команд или профилей работы. Реже ограничиваются одним предварительным интервью. Так происходит, если кандидат знаком интервьюерам или недавно проходил собеседование на другую вакансию.

Инженер, проводящий собеседование, обычно просит написать небольшой фрагмент кода в специальном редакторе для совместной работы. Также задаются разнообразные вопросы, показывающие, какими технологиями вы владеете.

Затем вас приглашают в Сиэтл (или другое отделение, в которое вы подали документы). Вам предстоит пройти еще четыре или пять собеседований с одной или двумя командами, которые выбрали вас на основании резюме и телефонного интервью. Вам предложат написать программный код, чтобы другие интервьюеры смогли оценить ваши навыки. У каждого интервьюера свой профиль, поэтому вопросы могут сильно различаться. Интервьюер не может знать решение коллег до тех пор, пока не поставит собственную оценку, а обсуждения запрещены вплоть до собрания, на котором принимается решение о найме.

В собеседованиях участвует главный интервьюер (*bar raiser*). Он проходит специальную подготовку и может общаться с кандидатами вне пределов их группы, чтобы сбалансировать группу в целом. Если вопросы одного из интервьюеров оказываются более сложными, чем у остальных, скорее всего, это главный интервьюер. Он имеет огромный опыт в проведении собеседований и право вето в решении о найме. Впрочем, даже если собеседование с главным интервьюером оказывается более сложным, это не значит, что вы показываете более низкие результаты. Ваша результативность оценивается с учетом сложности, а не только по проценту правильных ответов.

Как только интервьюеры выставили оценки, они встречаются для обсуждения. На встрече принимается решение о найме.

В большинстве случаев специалисты по подбору кадров от Amazon превосходно выполняют свои обязанности, но и у них бывают задержки. Если вы не получили ответ в течение недели, вежливо напомните о себе по электронной почте.

ПОДГОТОВИТЬСЯ! Amazon — веб-ориентированная компания. Убедитесь, что вы готовы к вопросам о масштабировании систем. Специальная подготовка в области объектно-ориентированных систем для ответа на эти вопросы не нужна. За информацией обращайтесь к главе 9 «Масштабируемость и проектирование систем».

Кроме того, в Amazon любят задавать вопросы по объектно-ориентированному программированию. Прочитайте главу 7 «Объектно-ориентированное проектирование».

ОСОБЕННОСТИ Главный интервьюер привлекается из другой команды для обеспечения качества собеседования. Если вы хотите пройти собеседование и получить работу, необходимо произвести должное впечатление на главного интервьюера и менеджера по найму.

Amazon чаще экспериментирует с процессом подбора кадров, чем другие компании. Описанный процесс типичен, но из-за экспериментов Amazon он не может считаться универсальным.

Google

О собеседованиях в Google ходит много страшных слухов, но это только слухи. На самом деле собеседование в Google не так уж сильно отличается от собеседования в Microsoft или Amazon.

Прежде всего инженер Google побеседует с вами по телефону, так что ожидайте непростых технических вопросов. Эти вопросы могут включать в себя написание кода, иногда через совместную работу с документом. Обычно предварительные собеседования по телефону и в офисе компании проводятся по одним стандартам и включают похожие вопросы.

На личном собеседовании с вами будут общаться от четырех до шести интервьюеров, в том числе и неформальный интервьюер (lunch interviewer). Решение интервьюера является тайной, никто не знает, что думает его коллега. Вы можете быть уверены, что интервьюер дает независимую оценку. Неформальный интервьюер свое мнение не сообщает, поэтому в общении с ним можно задать честный вопрос.

У интервьюеров нет согласованной «структуры» или «системы» вопросов. Каждый интервьюер может проводить собеседование так, как считает нужным.

Затем результаты собеседования передаются в комитет по подбору персонала, инженеры и менеджеры которого принимают решения о приеме на работу. Обычно оцениваются четыре критерия (аналитические способности, навыки программирования, опыт и коммуникативные способности), по каждому из них выставляются оценки от 1.0 до 4.0. Интервьюеры обычно не участвуют в работе комитета по подбору персонала (а исключения могут объясняться чистой случайностью).

Решающую роль играют ваши оценки. Комитет хочет видеть ваши преимущества по отношению к другим кандидатам. Другими словами, оценки 3.6, 3.1, 3.1, 2.6 предпочтительнее, чем все 3.1.

Не обязательно становиться лучшим на каждом собеседовании, а предварительное собеседование по телефону обычно не является решающим фактором.

Если комитет по подбору персонала примет положительное решение, ваш пакет документов будет направлен в комитет, занимающийся назначением заработной платы, затем в исполнительный управляющий комитет. Принятие окончательного решения затягивается на несколько недель, поскольку Google имеет множество разных уровней и комитетов.

ПОДГОТОВИТЬСЯ! Google — веб-ориентированная компания. Убедитесь, что вы готовы к вопросам о масштабировании систем. Специальная подготовка в области объектно-ориентированных систем для ответа на эти вопросы не нужна. За информацией обращайтесь к главе 9 «Масштабируемость и проектирование систем».

Google уделяет особое внимание аналитическим (алгоритмическим) навыкам независимо от уровня опыта. Вы должны очень хорошо подготовиться к таким вопросам, даже если считаете, что ваш опыт вас выручит.

ОСОБЕННОСТИ Ваши интервьюеры не принимают решение о найме, их оценки передаются в комитет по подбору персонала. Комитет дает рекомендации, которые могут быть отвергнуты руководством Google (впрочем, это происходит очень редко).

Apple

Процесс собеседований в Apple наименее бюрократичен (как, впрочем, и сама компания). Интервьюеры будут оценивать прежде всего технические навыки, но для них очень важно ваше страстное желание занять должность и любовь к компании. Активная работа с Mac не является обязательным условием, но кандидат должен быть по крайней мере знаком с этой системой.

Интервью обычно начинается с телефонного звонка специалиста по подбору кадров — он должен оценить ваши базовые навыки. Затем происходит серия технических собеседований с членами команды.

После приглашения в кампус вас обычно встречает специалист, который вкратце расскажет вам о процедуре собеседования. Затем вас ждут 6–8 собеседований с участниками команды и ключевыми людьми, с которыми она будет сотрудничать.

Вам предстоят собеседования 1-на-1 и 2-на-1. Будьте готовы выступать за доской и убедитесь, что умеете четко формулировать свои мысли. Обед с возможным будущим начальником может показаться неформальным мероприятием, но и он является продолжением собеседования. Каждый интервьюер занимается собственной областью и обычно не обсуждает ваши результаты с другими.

К концу дня интервьюеры сравнивают свои заметки. Если все считают вас приемлемым кандидатом, вас направляют на собеседование с директором и вице-президентом организации, в которую вы собираетесь поступать. Данное решение принимается достаточно неформально, но если вы доберетесь до этого этапа — это очень хороший признак. С другой стороны, если решение будет отрицательным, вас просто проведут на выход, и вы ничего не узнаете.

Если вы добрались до собеседования с директором и вице-президентом, все интервьюеры соберутся в конференц-зале, чтобы выразить свое официальное одобрение или неодобрение. Вице-президент на встрече обычно не присутствует, но может наложить вето, если вы произвели неблагоприятное впечатление. Обычно специалист

по подбору кадров связывается с вами через несколько дней, но ничто не мешает вам обратиться за информацией.

ПОДГОТОВИТЬСЯ! Если вы знаете, какая команда будет проводить собеседование, убедитесь, что знакомы с продуктом этой команды. Что вам в нем нравится? Что вы хотели бы улучшить? Продемонстрируйте свою заинтересованность.

ОСОБЕННОСТИ Интервью часто проводятся в режиме 2-на-1; не беспокойтесь, он принципиально не отличается от режима 1-на-1.

Не забывайте, что сотрудники Apple должны быть энтузиастами своей продукции, поэтому продемонстрируйте свой интерес.

Facebook

Отобранные кандидаты обычно проходят одно или два предварительных собеседования по телефону. Такие собеседования обычно носят технический характер и включают программирование (обычно в совместном редакторе документов).

После предварительного собеседования(-ий) вам будет предложено выполнить домашнее задание, объединяющее программирование с алгоритмическими задачами. Уделите особое внимание стилю программирования. Если вы никогда не работали в среде с рецензированием кода, будет полезно обратиться к коллеге, который сможет провести рецензирование.

Во время личного собеседования вам в основном придется общаться с техническими специалистами, но в собеседованиях также могут участвовать менеджеры по подбору кадров. Все интервьюеры прошли полноценное обучение, поэтому шансы на получение работы никак не зависят от выбора конкретного интервьюера.

Каждому интервьюеру в ходе личных собеседований назначается «роль», которая гарантирует отсутствие повторяющихся вопросов и формирование целостного представления о кандидате. Основные роли:

- **Поведенческая** («джедай»): в ходе собеседования оценивается ваша способность достичь успеха в среде Facebook. Насколько хорошо вы уживетесь с корпоративной культурой и ее ценностями? Что вас интересует? Как вы будете справляться с проблемами? Также приготовьтесь поговорить о своем интересе к Facebook — здесь нужны энтузиасты. Также на собеседовании могут задаваться вопросы по программированию.
- **Программирование и алгоритмы** («ниндзя»): стандартные вопросы по программированию и алгоритмам вроде тех, которые представлены в книге. Вопросы намеренно сделаны сложными. Вы можете использовать любой язык программирования на свое усмотрение.
- **Проектирование/архитектура** («пират»): кандидатам на должности, связанные с разработкой внутренних подсистем, могут задаваться вопросы по системному проектированию. Кандидатам на разработку интерфейсной части будут задаваться вопросы, связанные с этой дисциплиной. Открыто обсуждайте разные решения с присущими им достоинствами и недостатками.

Обычно следует ожидать двух собеседований типа «ниндзя» и одного типа «джедай». Опытные кандидаты также обычно проходят «пиратское» собеседование.

Интервьюеры пишут отзывы до того, как обсуждать ваши достижения с другими. Это гарантирует, что успех (или, наоборот, неудача) на одном из собеседований никак не повлияет на результаты следующего.

Как только все составят свои отзывы, заинтересованная в вас команда и менеджер по подбору персонала собираются для принятия окончательного решения. Затем рекомендации передаются в комитет по подбору персонала.

В Facebook ищут настоящих «камикадзе», способных докапываться до истины и разрабатывать масштабируемые решения на любом языке программирования. Знание PHP не играет ключевой роли, поскольку Facebook также требуются программисты на C++, Python, Erlang и других языках программирования.

ПОДГОТОВИТЬСЯ! Самой молодой из «элитных» IT-компаний нужны разработчики с предпринимательской жилкой. Покажите, что вы инициативны и можете быстро работать.

От вас ждут, что вы можете построить элегантное и масштабируемое решение на любом языке по вашему выбору. Знание PHP особой роли не играет, особенно если учесть, что во внутренней работе Facebook широко применяются C++, Python, Erlang и другие языки.

ОСОБЕННОСТИ В Facebook собеседование организуется в целом для всей компании, а не для какой-то конкретной команды. Если вас взяли на работу, вам предстоит пройти 6-недельный «курс молодого бойца», цель которого — помочь разобраться в огромной кодовой базе. Вы получите задания от старших разработчиков, изучите новые методы и, в конечном счете, получите большую свободу выбора проекта (чем при назначении проекта по результатам собеседования).

Palantir

В отличие от некоторых компаний, проводящих «общие» собеседования (когда собеседование ведется для компании в целом, а не для конкретной команды), Palantir использует собеседования в интересах конкретных команд. Иногда ваша заявка может быть перенаправлена в другую команду, для которой вы лучше подходите.

Отбор кандидатов для Palantir обычно начинается с двух предварительных собеседований по телефону. Такие собеседования продолжаются от 30 до 45 минут и имеют в основном технический характер. Вероятно, вам придется рассказать о своей предшествующей работе, а особое внимание в процессе общения будет уделяться алгоритмам.

Возможно, вам также будет отправлено задание HackerRank, которое оценит вашу способность к написанию оптимальных алгоритмов и правильного кода. Такие проверки с большей вероятностью достанутся менее опытным кандидатам, например студентам.

После этого успешные кандидаты приглашаются для личной встречи и проходят до пяти собеседований. В таких собеседованиях проверяется опыт работы, знание предметной области, структур данных и алгоритмов, а также навыки проектирования.

Скорее всего, вам покажут продукты Palantir. Задайте уместные вопросы и продемонстрируйте свой интерес к компании.

После собеседования интервьюеры встречаются для обсуждения результатов с менеджером по подбору персонала.

ПОДГОТОВИТЬСЯ! Palantir ценит одаренных технических специалистов. Многие кандидаты сообщают, что вопросы Palantir были труднее, чем в Google и других ведущих компаниях. Это не означает, что вам будет сложнее получить работу (хотя и это не исключено); просто интервьюер предпочитает более сложные вопросы. Если вы собираетесь проходить собеседование в Palantir, постарайтесь досконально изучить основные структуры данных и алгоритмы. Затем сосредоточьтесь на подготовке самых сложных вопросов по теме алгоритмов.

Также стоит освежить в памяти системное проектирование, если вы проходите собеседование на роль специалиста по архитектуре, — это важная часть процесса.

ОСОБЕННОСТИ Практическое программирование — стандартная часть собеседований в Palantir. Хотя вы будете сидеть за своим компьютером и можете работать со справочными материалами, не беритесь за работу без подготовки. Вопросы могут быть в высшей степени сложными, а эффективность предложенных вами алгоритмов будет тщательно анализироваться. Обстоятельная подготовка к собеседованию поможет вам справиться с заданием.

Также вы можете попрактиковаться в решении задач по программированию на сайте *HackerRank.com*.

III

Нестандартные случаи

Есть много причин, по которым люди берутся за подобные книги. Возможно, вам хватает опыта работы, но вы никогда не проходили подобные собеседования, а может, вы занимались тестированием или управлением проектами. Или вам хочется узнать из книги, как лучше организовать собеседование. В этом разделе каждый из этих «нестандартных случаев» найдет что-то полезное для себя.

Профессионал

Иногда приходится слышать мнение, что вопросы по алгоритмам, приведенные в книге, предназначены только для недавних выпускников. Это не совсем так. Возможно, кандидатам, имеющим профессиональный опыт, вопросы по алгоритмам задаются реже — но ненамного.

Если компания задает вопросы по алгоритмам неопытным кандидатам, обычно эти же вопросы предлагаются и кандидатам с опытом работы. Такие компании считают (правильно или ошибочно), что навыки, продемонстрированные в ответах на такие вопросы, важны для любого разработчика.

Некоторые интервьюеры немного «снижают планку» для опытных кандидатов. В конце концов, прошло много лет с тех пор, как кандидат слушал лекции по алгоритмам. Материал мог подзабыться.

Другие интервьюеры имеют противоположное мнение: они считают, что за годы практической работы кандидат видел много подобных задач.

В среднем же подходы различных интервьюеров более или менее сбалансированы. Исключением из этого правила являются вопросы по системному проектированию и архитектуре, а также вопросы по резюме. Обычно студенты не занимаются системной архитектурой, поэтому знания они получают только во время работы. Следовательно, ваш результат на собеседовании по этой теме будет оцениваться с учетом вашего практического опыта. Впрочем, студентам и недавним выпускникам также задают эти вопросы, и они должны быть готовы ответить на них так, как смогут.

Также ожидается, что опыт кандидата позволит ему дать более глубокий и впечатляющий ответ на вопросы типа «Расскажите о самой серьезной ошибке, с которой вы столкнулись в работе». У вас больше опыта, и вы должны продемонстрировать это в своем ответе.

Тестеры и SDET

Специалисты SDET (Software Design Engineer in Test, программист-тестер) тоже пишут код — но не для реализации функциональности, а для ее тестирования.

Соответственно, они должны быть не только прекрасными программистами, но и прекрасными тестерами. Требования возрастают вдвое!

Если вы претендуете на должность тест-программиста:

- Подготовьтесь к базовым задачам тестирования. Как вы будете тестировать лампочку? Ручку? Кассовый аппарат? Microsoft Word? Глава, посвященная тестированию, поможет подготовиться к вопросам такого рода.
- Подготовьтесь к заданиям, связанным с программированием. Главная причина для отказов на должности SDET — недостаточная квалификация программиста. Хотя требования к SDET ниже, чем для рядового разработчика, вы должны разбираться в алгоритмах. Убедитесь, что вы можете справиться с заданиями по алгоритмизации и программированию, которые даются обычным разработчикам.
- Отработайте навыки тестирования вопросов по программированию. В качестве примера возьмем очень популярное задание: «Напишите код, который делает X», а затем: «Хорошо, а теперь протестируйте его». Даже если в вопросе об этом явно не сказано, спросите себя: «Как бы я протестировал этот код?» Помните: любое задание может превратиться в задачу по тестированию.

Для тестеров программного обеспечения очень важны коммуникативные навыки, поскольку им приходится работать с людьми. Не игнорируйте часть V «Подготовка к поведенческим вопросам».

Совет

В завершение этого раздела хочу дать вам небольшой совет, который положительно отразится на вашей карьере. Если вы, как и многие другие кандидаты, расцениваете должность тестера как хорошую «ступеньку» для вхождения в компанию, не обольщайтесь. Для многих кандидатов переход с этой должности на должность разработчика оказался трудным испытанием. Если вы все-таки решились на этот шаг, убедитесь в том, что вы сохранили все свои навыки в области программирования и алгоритмов, и постарайтесь осуществить переход в течение одного-двух лет. Чем больше пройдет времени, тем меньше вероятность того, что вас будут серьезно воспринимать на собеседовании на должность разработчика.

Поддерживайте свои навыки программирования на должном уровне.

Менеджеры программ и менеджеры продукта

PM[1] — общая аббревиатура, которой обозначают как менеджеров программ, так и менеджеров продукта. Но роли и задачи этих двух PM сильно различаются даже в пределах одной компании. В Microsoft, например, некоторые PM фактически выполняют функции маркетологов, то есть больше общаются с клиентами,

[1] В крупных компаниях над одним и тем же продуктом, помимо рядовых программистов и тестировщиков, работают менеджеры. Менеджер управления программой (Program Manager) отвечает за архитектуру решения, а вот менеджер управления продуктом (Product Manager) занимается маркетингом и представляет интересы заказчика. Получается, что первый — программист, а второй — маркетолог. — *Примеч. пер.*

нежели программируют, а вот в других компаниях PM могут провести большую часть своего рабочего дня, занимаясь программированием. Таким кандидатам на собеседованиях скорее достанутся вопросы по программированию, потому что это важная часть их работы.

Когда интервьюеры ищут кандидата на должность PM, они ищут человека, способного продемонстрировать следующие навыки:

- *Обработка неоднозначностей.* Это не самая главная часть собеседования, но вы должны знать, что интервьюеры приветствуют подобные навыки. Им важно понять, что вы будете делать, когда столкнетесь с неоднозначной ситуацией. Постарайтесь продемонстрировать, что вы не остановитесь. Интервьюер хочет видеть, что вы активно беретесь за решение: ищете информацию, выделяете наиболее приоритетные составляющие и организованно подходите к решению задачи. Обычно такие навыки не проверяются напрямую (хотя и такое возможно), но они могут стать одним из аспектов, на которые обратит внимание интервьюер.
- *Ориентированность на потребителя (подход).* Интервьюеры хотят видеть, что вы ориентируетесь на целевую аудиторию. Вы твердо уверены, что все потенциальные потребители будут использовать программный продукт точно так же, как и вы? Или вы способны взглянуть на продукт с точки зрения клиента и попытаетесь понять, как он будет пользоваться программой? Вопросы типа «Разработайте будильник для слепого» позволяют изучить этот аспект. Столкнувшись с подобным вопросом, непременно уточните, *кто* является вашей целевой аудиторией и *как* он будет использовать продукт. Необходимые навыки описаны в главе 11 «Тестирование».
- *Ориентированность на потребителя (технические навыки).* Некоторые команды, работающие со сложными продуктами, должны быть уверены в том, что их PM хорошо понимают сам продукт. Трудно приобрести такое понимание во время работы. Возможно, глубокие технические знания в области мобильных телефонов не обязательны для работы в группах Android или Windows Phone (хотя они и могут пригодиться), но понимание задач безопасности является необходимым условием для работы над Windows Security. Чтобы пройти собеседование на должность, требующую определенных технических навыков, необходимо по крайней мере заявить о наличии у вас таких навыков.
- *Многоуровневые коммуникации.* PM должен уметь общаться с сотрудниками компании всех уровней подготовки и любого статуса. Ваш интервьюер захочет убедиться, что вы обладаете подобной способностью. Это легко проверяется, например, с помощью простого задания «Объясните, что такое TCP/IP, своей бабушке». Ваши коммуникативные способности также могут оцениваться по рассказу о вашей предыдущей работе.
- *Страсть к новым технологиям.* Счастливые сотрудники — это продуктивные сотрудники, поэтому компания должна убедиться, что вы будете получать удовольствие от работы. В ваших ответах должно отразиться ваше отношение к новым технологиям — а в идеале и к конкретной команде или компании. Например, вас могут напрямую спросить: «Чем вам привлекает Microsoft?» Интервьюеры ожидают увидеть энтузиазм в вашем рассказе о предшествующем

опыте и задачах команды. Они хотят видеть, что вы получаете удовольствие от решения сложных задач.

- *Работа в команде/лидерство.* Это может стать решающим моментом собеседования — ведь это и есть ваша работа. Все интервьюеры пытаются оценить то, как хорошо вы работаете с другими людьми. Чаще всего для этого используются вопросы типа «Расскажите мне о ситуации, когда ваш член команды не выполнял свою часть обязанностей». Интервьюер смотрит, способны ли вы справляться с конфликтами, берете ли на себя инициативу, понимаете ли вы коллектив и нравится ли людям работать с вами. Вам следует уделить должное внимание поведенческим вопросам.

Все перечисленные навыки важны для PM и являются ключевыми для собеседования. Внимание, уделяемое каждой области в собеседовании, примерно соответствует ее важности в реальной работе.

Ведущие разработчики и менеджеры

Отличные навыки программирования являются необходимым условием для ведущих разработчиков и очень часто требуются и от «управленцев». Если ваша работа будет связана с программированием, убедитесь, что ваши познания в алгоритмизации и программировании находятся на достаточно высоком уровне. В частности, Google предъявляет к менеджерам высокие требования в том, что касается программирования.

Дополнительно вас будут оценивать по следующим критериям:

- *Работа в команде/лидерство.* Претендент на любую руководящую должность обязан быть лидером и уметь работать с людьми. На интервью вас будут оценивать явно и неявно. При открытой оценке вам задают вопрос: «Что вы будете делать, если не согласны с менеджером?» Неявная оценка проводится по результатам вашего общения с интервьюерами. Если интервьюер увидит, что вы высокомерны или, наоборот, слишком пассивны, он решит, что менеджер из вас не получится.
- *Расстановка приоритетов.* Менеджеры часто сталкиваются с непростыми проблемами — например, им приходится следить за тем, чтобы команда укладывалась в жесткие сроки. Ваши интервьюеры хотят видеть, что вы способны правильно расставить приоритеты и отбросить все второстепенное. Кроме того, вы должны уметь задавать правильные вопросы, чтобы понять, что действительно критично и на что можно реально рассчитывать.
- *Коммуникативные навыки.* Менеджерам приходится много общаться с людьми, стоящими как выше, так и ниже их по карьерной лестнице. Они должны общаться с потенциальными клиентами и менее технически подкованными людьми. Интервьюеры хотят видеть, что вы способны общаться с разными людьми, оставаясь дружелюбными и доброжелательными. По сути, на собеседовании оценивается ваша личность.
- *Доведение дела до конца.* Менеджер должен быть человеком, который всегда добивается своей цели. Вам необходимо выдержать баланс между подготовкой к проекту и его реализацией. Менеджер должен знать, как структурировать проект и как мотивировать людей для достижения целей команды.

В конечном счете, все эти критерии имеют отношение к вашему предыдущему опыту и к вашей личности. Очень, очень хорошо подготовьтесь с использованием таблицы подготовки к собеседованию.

Стартапы

В стартапах процессы подачи заявления и собеседования существенно различаются в зависимости от конкретной фирмы. Мы не можем описать специфику каждого стартапа, но можем обсудить некоторые общие черты. Помните, что у каждого стартапа могут быть свои особенности, отклоняющиеся от общей схемы.

Процесс подачи заявления

Многие стартапы публикуют списки вакансий, но часто лучшим способом устройства на работу является личная рекомендация. Приглашающий не обязательно будет вашим коллегой или другом. Часто достаточно просто выразить свой интерес, а кто-нибудь увидит ваше резюме и решит, что вы можете оказаться хорошим кандидатом.

Виза и разрешение на работу

К сожалению, небольшие стартапы в США не способны предоставить рабочую визу для своих сотрудников. Поверьте, они недовольны подобной ситуацией, но ничем не могут помочь. Если вам нужна рабочая виза, лучше всего обратиться к профессиональному вербовщику, который работает со многими стартапами (и лучше представляет, какие стартапы готовы заниматься визами), или поискать более крупную компанию.

Резюме

Стартапы обычно привлекают людей, которые не только являются техническими специалистами, но и обладают предпринимательскими наклонностями. В идеале в вашем резюме должна быть отражена ваша инициативность. Какие проекты вы запускали?

Еще один важный аспект — возможность сходу взяться за работу. В стартапах ищут людей, уже знакомых с языком программирования, который используется в данной компании.

Процесс собеседования

В отличие от крупных компаний, которые оценивают ваше умение разрабатывать программное обеспечение, стартапы в основном смотрят на ваши личностные качества, квалификацию и предшествующий опыт:

- *Личная совместимость.* Личная совместимость обычно оценивается по вашему общению с интервьюером. Дружеская, содержательная беседа — это залог получения вакансии.
- *Набор навыков.* Поскольку стартапам нужны люди, которые сразу могут приступить к работе, они будут оценивать ваши навыки программирования на конкретном языке. Если вы знакомы с языком, на котором работает стартап, повторите его основы.

- *Предыдущий опыт.* В стартапах задают много вопросов о предыдущем опыте. Уделите особое внимание части V «Подготовка к поведенческим вопросам».

В дополнение к перечисленным критериям на собеседовании часто задаются вопросы по программированию и алгоритмизации, которые вы найдете в этой книге.

Для интервьюеров

С момента выхода последнего издания я узнала, что многие интервьюеры используют книгу для того, чтобы научиться проводить собеседования. Книга была написана не для этого, но я также приведу несколько советов для интервьюеров.

Не задавайте те вопросы, которые приведены в книге

Во-первых, эти вопросы были выбраны потому, что они хорошо подходят для подготовки к собеседованиям. Такие вопросы не всегда хорошо подходят для самих собеседований. Например, в книге представлены вопросы-«головоломки», потому что некоторые интервьюеры задают такие вопросы. Кандидатам стоит потренироваться в их решении на случай, если они будут проходить собеседование в такой компании, хотя лично я считаю, что такие вопросы плохо годятся для собеседований.

Во-вторых, ваши кандидаты тоже читают книги. Не стоит задавать вопросы, которые могут быть известны кандидату.

Задавайте *похожие* вопросы, но не копируйте их от и до. Ваша цель — проверить умение решать задачи, а не память кандидата.

Предлагайте задачи средней и высокой сложности

Цель этих вопросов — оценка навыков решения задач. Если вы задаете слишком простые вопросы, эффективность собеседования снижается, а кандидат отвлекается на второстепенные аспекты.

Ищите вопросы с несколькими скрытыми затруднениями

Некоторые вопросы сильно зависят от моментов озарения. Если счастливая догадка не посетит кандидата, он плохо справится с заданием, а если посетит — он обойдет многих других кандидатов. Даже если озарение может считаться индикатором навыков, это всего лишь один из возможных индикаторов. В идеале следует искать задачи с несколькими препятствиями, догадками или оптимизациями. Множественные показатели лучше одиночных.

Полезный критерий: если вы можете дать подсказку или совет, способный существенно изменить результаты кандидата, то, скорее всего, это не лучший вопрос для собеседования.

Не требуйте феноменальной эрудиции

Некоторые интервьюеры, стремящиеся усложнить вопросы, неспециально завышают требования к знаниям кандидата. Конечно, чем меньше кандидатов успешно справляется с заданиями, тем более впечатляюще выглядит статистика, но этот факт мало что говорит о реальных навыках кандидатов.

От кандидатов следует ожидать разве что знания относительно простых структур данных и алгоритмов. Разумно предположить, что недавний выпускник понимает основы синтаксиса «О» большого и деревьев, но мало кто вспомнит алгоритм Дейкстры или тонкости работы деревьев АВЛ.

Если вопрос собеседования требует экзотических познаний, спросите себя: действительно ли этот навык настолько важен? Он настолько важен, что я предпочту сократить количество подходящих кандидатов или же уделять меньше внимания навыкам практического решения задач?

Каждый новый навык или атрибут, проверяемый вами, сокращает количество потенциальных кандидатов, если только повышенные требования не компенсируются ослаблением требований к другому навыку. Конечно, при всех равных условиях вы предпочтете кандидата, способного наизусть цитировать толстенный учебник по алгоритмам. Только в данном случае не все условия равны.

Избегайте «пугающих» вопросов

Некоторые вопросы страшат кандидатов, потому что на первый взгляд они требуют специальных познаний (даже если в действительности это не так). К этой категории часто относятся вопросы, в которых задействованы:

- математика и теория вероятностей;
- низкоуровневые функции (выделение памяти и т. д.);
- проектирование или масштабируемость систем;
- фирменные системы (Google Maps и т. д.).

Например, на собеседованиях я иногда предлагаю найти все положительные целочисленные решения уравнения $a^3 + b^3 = c^3 + d^3$, меньшие 1000. Многие кандидаты думают, что им придется заниматься хитроумным разложением на множители или какими-то нетривиальными вычислениями. На самом деле это не так. Кандидат должен понимать концепцию возведения в степень, суммы и равенства — и это все.

Задавая этот вопрос, я явно сообщаю: «Может показаться, что это математическая задача. Не беспокойтесь, это вопрос на алгоритмы». Если кандидат берется за разложение, я останавливаю его и напоминаю, что эта задача не на математику.

В других вопросах может быть задействована теория вероятностей. Часто речь идет о тривиальных вещах, наверняка известных кандидату (скажем, случайный выбор одного из пяти вариантов или выбор случайного числа от 1 до 5). Но сам факт того, что в задаче задействованы вероятности, пугает кандидатов.

Будьте внимательны с вопросами, которые могут показаться устрашающими. Помните, что кандидат и так нервничает. Добавление «пугающего» вопроса может окончательно нарушить его душевное равновесие, в результате чего он выступит хуже своего реального уровня.

Если вы собираетесь задать вопрос, который кажется «пугающим», обязательно заверьте кандидата, что вопрос не требует каких-то особых знаний.

Обеспечьте позитивный настрой

Некоторые интервьюеры уделяют столько внимания подбору «правильных» вопросов, что забывают думать о своем собственном поведении.

Многие кандидаты нервничают на собеседованиях и прислушиваются к каждому слову интервьюера. Они улавливают все, что может показаться позитивным или негативным, и считают, что простое пожелание «Удачи!» несет какой-то смысл, даже если вы говорите это всем независимо от показанных результатов.

Кандидаты не должны переживать относительно вас и хода собеседования. Вы должны успокоить их. Нервничающий кандидат покажет плохие результаты, но это совершенно не означает, что он недостаточно хорош. Более того, хороший кандидат с негативной реакцией на вас или на компанию с меньшей вероятностью примет предложение — и может отговорить своих знакомых от собеседования или сделанного предложения.

Постарайтесь проявить теплоту и дружелюбие. У одних людей это получается лучше, у других хуже, и все же постарайтесь. Даже если вам это дается нелегко, постарайтесь в ходе собеседования почаще произносить ободряющие слова:

- «Верно, именно так».
- «Точно подмечено».
- «Превосходно».
- «Очень хорошо, интересная мысль».

Как бы плохо ни шли дела у кандидата, хоть что-то он делает правильно. Найдите возможность добавить позитива в собеседование.

Уделяйте больше внимания поведенческим вопросам

Многие кандидаты не умеют описывать свои достижения. Вы задаете им вопросы о сложной ситуации, а они рассказывают о сложной ситуации, с которой столкнулась их команда. Насколько можно судить по их словам, сами кандидаты особого участия в этом не принимали.

Однако не стоит торопиться. Может оказаться, что кандидат не говорит о себе, потому что его учили не хвалиться своими достижениями, а концентрироваться на достижениях команды. Это относится в первую очередь к кандидатам на руководящие должности и женщинам.

Не стоит полагать, что кандидат в некоторой ситуации ничего не сделал, только потому, что вы не поняли, что же именно было сделано. Привлеките внимание кандидата к ситуации (деликатно!). Напрямую спросите, сможет ли он описать свою роль.

Если из сказанного не создается впечатления, что ситуация была достаточно сложной, ищите глубже. Попросите кандидата более подробно рассказать, в чем, по его мнению, заключалась проблема и какие действия они предприняли. Спросите, почему были предприняты те или иные действия.

Участвовать в собеседовании в роли кандидата тоже нужно уметь (в конце концов, именно поэтому была написана книга). Однако это не тот навык, который вам хотелось бы проверять.

Если кандидату нужно время — дайте ему время

Многие кандидаты спрашивают меня: что делать, если интервьюер требует отвечать, когда кандидату нужно время для размышлений?

Если кандидату нужно время, предоставьте ему время для размышлений. Научитесь различать «Я в тупике и понятия не имею, что делать» и «Я размышляю в тишине». Возможно, содействие со стороны интервьюера поможет многим кандидатам, но не обязательно всем. Некоторым нужно подумать. Дайте им это время для размышлений и учтите в своих оценках, что эти кандидаты получили от вас меньше информации, чем другие.

Выбор категории

На очень высоком уровне вопросы можно разделить на четыре категории:

Базовые вопросы: часто это простые вопросы на решение задач или проектирование, оценивающие минимальную компетентность кандидата. По их результатам отличить «неплохо» от «замечательно» не удастся, даже не пытайтесь. Используйте эти вопросы на ранней стадии процесса (чтобы отфильтровать самых неподходящих кандидатов) или в тех случаях, когда для вакансии достаточно минимальной квалификации.

Расширенные вопросы: более сложные вопросы, часто также связанные с решением задач или проектированием. Это серьезные вопросы, над которыми кандидат должен как следует подумать. Используйте их в том случае, если навыки алгоритмизации и решения задач особенно важны.

Специализированные вопросы: проверка знаний в узкоспециализированных областях (например, язык Java или машинное обучение). Обычно используются для тех областей, которые хороший инженер не сможет быстро освоить во время работы. Вопросы должны быть уместными для настоящего специалиста. К сожалению, мне приходилось видеть, как кандидату после 10-недельных курсов программирования задавались сложные вопросы по Java. О чем это говорит? Если кандидат знает ответы, то он узнал их недавно, а следовательно, эти знания приобретаются легко, а если они приобретаются легко, то нет смысла искать кандидата с познаниями в этой области.

Дополнительные знания: эти вопросы не относятся к уровню специализированных. Собственно, для вакансии эти знания даже могут быть не нужны, но разумно ожидать, что специалист такого уровня владеет этой информацией. Например, для вас может быть несущественно, владеет кандидат CSS или HTML, но если кандидат серьезно работал с этими технологиями, но не может объяснить, для каких целей хорошо (или плохо) подходят таблицы, это выглядит странно. Получается, что кандидат не усваивает информацию, которая играет важную роль в его работе.

Проблемы начинаются тогда, когда интервьюеры смешивают эти категории:

- Они задают вопросы для специалистов людям, которые специалистами не являются.
- Они открывают вакансии для специалистов тогда, когда эти специалисты им не нужны.

- Им нужны специалисты, но на собеседованиях оцениваются только базовые навыки.
- Они задают базовые вопросы, считая их расширенными. По этой причине они уделяют большое внимание различиям между «приемлемым» и «отличным» результатом, хотя эти различия могут быть совершенно незначительными.

Мой опыт участия в процессе найма многих мелких и крупных компаний показал, что эти ошибки встречаются гораздо чаще, чем можно было бы ожидать.

IV

Перед собеседованием

Успешное прохождение собеседования начинается задолго до самого собеседования — можно сказать, за годы. Если вы подключаетесь к этому процессу на поздней стадии, не беспокойтесь. Постарайтесь наверстать упущенное, насколько это возможно, а потом сосредоточьтесь на подготовке.[1] Удачи!

Получаем «правильный» опыт

Без хорошего резюме нет собеседования, а без предыдущего опыта не будет хорошего резюме. Следовательно, начать следует с приобретения опыта. Чем дальше будут простираться ваши планы на будущее, тем лучше.

Для студентов это может означать следующее:

- *Принимайте участие в больших проектах.* Ищите учебные курсы, участвующие в больших проектах. Это хороший способ получить практический опыт еще до того, как вы формально приступите к работе. Чем больше проект связан с реальными задачами, тем лучше.
- *Устройтесь на стажировку.* Сделайте все, чтобы попасть на стажировку на ранней стадии обучения. Это откроет путь к еще более полезной стажировке перед получением диплома. Во многих ведущих технических компаниях предусмотрены программы стажировки, предназначенные специально для новичков и студентов-второкурсников. Если вы не можете принять участие в подобной программе, найдите стартап и попробуйте свои силы.
- *Попробуйте начать какой-либо проект.* Начните проект в свободное время, участвуйте в марафонах по программированию или участвуйте в проекте с открытым кодом. Такая работа не только увеличит ваши технические навыки и практический опыт, но и произведет впечатление на компанию.

С другой стороны, профессионал может уже обладать опытом, который позволит ему попасть в компанию его мечты. Например, у разработчика в Google может быть достаточно опыта для перехода в Facebook. Впрочем, если вы пытаетесь перейти из менее известной компании в один из «флагманов» или переключиться с тестирования на разработку, вам пригодятся следующие советы:

- *Сделайте так, чтобы ваши рабочие обязанности максимально приблизились к задачам программирования.* Не показывайте своему руководству, что вы думаете об уходе, но уделите больше внимания программированию. По возможности выбирайте «содержательные» проекты и используйте актуальные технологии — это

[1] Схему подготовки к собеседованию скачайте с сайта издательства http://goo.gl/ssQdRk

станет дополнительным плюсом для вашего резюме (в идеале такие проекты сформируют его основу).

- *Используйте все свободное время — ночи и выходные дни.* Если у вас появилось несколько минут или часов, займитесь разработкой любого приложения — мобильного, настольного или веб-приложения. Такие проекты помогут вам освоить новые технологии и повысят вашу ценность для современных компаний. Обязательно упомяните эти проекты в резюме: люди, разрабатывающие программы «ради собственного удовольствия», производят хорошее впечатление на интервьюеров.

Все это сводится к двум основным моментам, которые хотят увидеть компании: что вы умны и что вы умеете программировать. Если вы можете это доказать, у вас больше шансов пройти собеседование.

Кроме того, нужно заранее думать о развитии вашей карьеры — какой дорогой она должна пойти? Если вы собираетесь сделать карьеру в области руководства, даже если в настоящий момент вы ищете вакансию разработчика, постарайтесь набрать опыт руководящей работы.

Идеальное резюме

Аналитики, просматривающие резюме, обращают внимание на те же самые детали, что и интервьюеры. Они хотят понять, насколько вы умны и умеете ли программировать.

Это означает, что вам нужно сделать акцент в резюме на «правильных» данных, и ваша любовь к теннису, путешествиям или карточным фокусам ничего не сделает в этом направлении. Подумайте, стоит ли сокращать информацию технического характера, заменяя ее описанием разнообразных увлечений.

Правильный размер

Резюме должно укладываться в одну страницу, если ваш опыт работы не превышает десяти лет, или две страницы, если вы более опытны. Впрочем, у многих опытных кандидатов размер резюме в 1,5–2 страницы может оказаться оправданным.

Дважды подумайте, прежде чем писать длинное резюме. Короткие резюме часто производят большее впечатление.

- *Рекрутеры тратят на одно резюме в среднем около 10 секунд.* Если вы сократите размер и укажете в резюме только самые заметные детали, их наверняка заметят. Обилие информации только отвлекает рекрутера от того, на что следовало бы обратить его внимание.
- *Некоторые люди просто отказываются читать длинные резюме.* Вы же не хотите, чтобы ваше резюме было отклонено?

Если вы думаете, что ваш обширный опыт невозможно описать на одной странице, поверьте мне — это *возможно*. Длинное резюме еще не является доказательством опытности претендента. Оно говорит лишь о том, что вы не можете правильно расставить приоритеты при его написании.

Трудовой стаж

Ваше резюме не должно включать полную историю всех должностей, которые вы когда-либо занимали. Включайте только значимые позиции — такие, которые произведут впечатление на будущего работодателя.

Указывайте только значимые позиции

Для каждой занимаемой вами должности необходимо добавить описание достижений: «При осуществлении X я добился Y, что привело к Z». Пара примеров:

- «Благодаря моей реализации распределенного кэширования время прорисовки сократилось на 75 %, что привело к сокращению времени входа в систему на 10 %».
- «Благодаря реализации нового алгоритма сравнения на базе windiff средняя точность совпадений выросла с 1,2 до 1,5».

Конечно, не нужно пытаться формализовать все ваши достижения, но принцип, думаю, ясен. Покажите, что вы сделали, как вы это сделали и какие результаты получены. В идеале постарайтесь найти для ваших результатов количественное выражение.

Проекты

Раздел «Проекты» в вашем резюме — лучший способ продемонстрировать свой опыт. Наиболее важно это для учащихся или недавних выпускников.

В список нужно включать два–четыре самых существенных проекта. Опишите проект: на каком языке он был реализован, какие технологии были использованы. Необходимо упомянуть, был ли проект индивидуальным или над ним работала целая команда. Все эти детали необязательны, поэтому включайте их только в том случае, если вы от этого предстанете в лучшем свете. Обычно индивидуальные проекты ценнее коллективных, так как они демонстрируют инициативность кандидата.

Не добавляйте слишком много проектов. Многие кандидаты делают ошибку, перечисляя всё, чем когда-либо занимались, забивая свое резюме небольшими, не впечатляющими проектами.

Языки программирования и программные продукты

Программные продукты

Проявите сдержанность при составлении списка программных продуктов, которыми вы владеете. Такие продукты, как Microsoft Office, почти всегда можно пропустить. Навыки работы с техническими продуктами (Visual Studio, Eclipse) более актуальны, они могут оказаться полезными, но для многих ведущих компаний не имеют особого значения. В конце концов, так ли уж трудно научиться работать в Visual Studio?

Конечно, вреда от такого списка не будет — просто он займет ценное место.

Языки программирования

Что перечислять? Все языки, на которых когда-либо вам приходилось программировать, или только те, которые часто используете?

Первый вариант слишком рискован. Многие интервьюеры считают, что вы готовы предоставить подробную информацию по любому пункту резюме. Я рекомендую перечислить большинство языков, которыми вы владеете, но обязательно укажите свой уровень опыта, например:

- Языки программирования: Java (эксперт), C++ (опытный), JavaScript (новичок).

Используйте те формулировки, которые наиболее адекватно описывают вашу квалификацию.

Некоторые разработчики указывают опыт работы на конкретном языке в годах, но от этой информации особой пользы нет. Если вы начали изучать Java 10 лет назад, но работали на этом языке от случая к случаю, можно ли это считать 10-летним опытом работы?

По этой причине стаж работы в годах — не лучшая метрика для резюме. Лучше просто изложите то, что хотите сказать, на обычном языке.

Если английский не ваш родной язык

Некоторые компании не станут рассматривать ваше резюме, если в нем будет много грамматических ошибок. Попросите кого-нибудь, для кого английский язык родной, проверить ваше резюме.

В резюме, отправляемом в американскую компанию, *не нужно* указывать возраст, семейное положение и национальность. Это личная информация, которая создает трудности для компании и возлагает на нее юридическую ответственность за конфиденциальное хранение и обработку ваших данных.

V

Подготовка к поведенческим вопросам

Поведенческие вопросы

Поведенческие вопросы задают, чтобы оценить вас как личность, уточнить резюме или просто снять напряжение в ходе собеседования. Так или иначе, эти вопросы важны и к ним следует подготовиться.

Как подготовиться

Пройдитесь по всем проектам и компонентам в своем резюме и убедитесь в том, что вы готовы подробно обсудить каждый пункт. Я рекомендую подготовить и заполнить специальную таблицу.

Частые вопросы	Проект 1	Проект 2	Проект 3
Проблемы, с которыми вы столкнулись			
Ошибки/неудачи			
Что понравилось			
Руководство			
Конфликты			
Что бы вы сделали иначе			

В шапке таблицы перечислите основные пункты вашего резюме: проекты, должности или другую деятельность.

Изучите таблицу перед собеседованием. Я рекомендую сократить каждую историю до пары ключевых слов, которые можно легко вписать в ячейку и легко вспомнить все остальное. Также такую таблицу можно будет держать перед глазами во время собеседования.

Также проследите за тем, чтобы было от одного до трех проектов, которые бы вы могли обсудить подробно, с анализом всех технических компонентов. Это должны быть проекты, в которых вам отводилась центральная роль.

Ваши слабые места

Когда вас спросят о слабых местах, расскажите о настоящих слабых местах. Ответы «Мое самое слабое место — я трудоголик», заставят интервьюера думать, что вы

слишком высокого мнения о себе или не хотите признаваться в своих слабостях. Лучше всего сказать правду, указать ваши подлинные слабые стороны, но продемонстрировать, как вы стараетесь преодолеть свои недостатки.

Например: «Я бываю не очень внимателен к деталям. В этом есть и хорошая сторона — я быстро выполняю задания, но иногда все-таки делаю ошибки по невнимательности. Именно поэтому я по несколько раз проверяю полученный результат».

Какие вопросы нужно задавать интервьюеру

Большинство интервьюеров дают вам шанс задать вопрос. Качество ваших вопросов, сознательно или неосознанно, повлияет на их решение. Приходя на собеседование, подготовьте такие вопросы заранее.

Вопросы можно разделить на три категории.

Настоящие вопросы

На эти вопросы вы, скорее всего, хотите получить ответы. Вот несколько вариантов, которые интересны многим кандидатам:

1. Какое количественное соотношение между тестерами, разработчиками и руководителями проектов? Как они взаимодействуют? Как происходит планирование проекта?
2. Что привело вас в эту компанию? С чем было больше всего проблем?

Эти вопросы помогут вам понять, как происходит ежедневная работа в компании.

Содержательные вопросы

Эти вопросы демонстрируют ваше знание и понимание технологий:

1. Я заметил, что вы используете технологию X. Как вы решаете проблему Y?
2. Почему продукт использует протокол X, а не Y? Я знаю, что такое решение обладает преимуществами A, B, C, но много компаний отказываются от него из-за проблемы D.

Чтобы задать такие вопросы, нужно заранее изучить продукты компании.

«Фанатские» вопросы

Эта категория вопросов позволяет продемонстрировать ваше отношение к конкретной технологии. Они показывают, что вы заинтересованы в обучении и компании:

1. Я очень интересуюсь темой масштабируемости. Какие возможности в этой области предоставит мне работа в вашей компании?
2. Я не знаком с технологией X, но слышал, что это очень интересное решение. Не могли бы вы мне рассказать, как она работает?

Знание технических проектов

В процессе подготовки следует сосредоточиться на двух-трех технических проектах, которые вы должны знать особенно глубоко. В идеале выбранные вами проекты должны удовлетворять следующим критериям:

- Наличие сложных компонентов (не просто «многому научился в ходе работы»).
- Вы сыграли центральную роль в работе (в идеале над сложными компонентами).
- Возможность обсуждения на глубоком техническом уровне.

Приготовьтесь обсудить проблемы, ошибки, технические решения, выбор технологий (их достоинства и недостатки), а также то, что вы бы сделали иначе. Также продумайте ответы на сопроводительные вопросы: например, как бы вы подошли к масштабированию приложения.

Ответы на поведенческие вопросы

Эта часть собеседования позволяет интервьюеру лучше узнать вас и оценить вашу компетентность. Запомните несколько советов — они пригодятся, когда вы будете отвечать на вопросы.

Отвечайте четко, но без высокомерия

Высокомерие — нехороший признак, но вы же хотите произвести впечатление. Как этого добиться и не показаться высокомерным? Будьте конкретными!

Излагайте факты и предоставьте интервьюеру возможность сделать выводы. Например, вместо того чтобы говорить «Именно я проделал самую сложную часть работы», опишите сложные компоненты.

Сократите подробности до минимума

Когда кандидат много и долго рассказывает о проекте, интервьюеру, который, может быть, не очень разбирается в предмете, трудно понять, о чем говорит кандидат.

Ограничьте подробности и оставьте только ключевые пункты. По возможности постарайтесь объяснить последствия. Интервьюер всегда сможет обратиться за более подробными сведениями.

Небольшой пример: «При исследовании типичного поведения пользователей я применял алгоритм Рабина—Карпа и самостоятельно разработал новый алгоритм, который позволил сократить время поиска с $O(n)$ до $O(\log n)$ в 90 % случаев. Могу рассказать подробнее, если вам это интересно».

Тем самым вы поясняете ключевые моменты и даете интервьюеру возможность получить дополнительную информацию.

Сосредоточьтесь на себе, а не на команде

Собеседование направлено на индивидуальную оценку. К сожалению, многие кандидаты (особенно претендующие на руководящие позиции) в своих ответах говорят «мы» и «команда». В итоге интервьюер плохо представляет, какой реальный вклад внес кандидат в общую работу, и может заключить, что этот вклад был незначительным.

Следите за своими ответами. Прислушивайтесь к тому, как часто вы говорите «мы» вместо «я». Помните, что каждый вопрос относится к вашей роли, и отвечайте с учетом этого.

Структурируйте ответ

Существует два способа дать структурированный ответ на поведенческий вопрос: «сначала суть» и SAR. Эти две техники можно использовать как по отдельности, так и вместе.

«Сначала суть»

Техника «золотой самородок» предполагает, что вы сразу выкладываете перед интервьюером краткое описание сути вашего ответа. Например:

- *Интервьюер*: «Расскажите, был ли в вашей карьере случай, когда вам приходилось убеждать людей внести значительные изменения?»
- *Кандидат*: «Несомненно. Позвольте мне рассказать, как я убедил администрацию колледжа разрешить студентам вести собственные курсы. Изначально в моей школе было правило, которое...»

Эта техника позволяет вам привлечь внимание интервьюера и предельно четко сообщить, о чем пойдет речь. Вам также будет проще сосредоточиться на изложении, так как вы уже представили «выжимку» своего ответа.

SAR

Метод SAR (Situation, Action, Result — ситуация, действие, результат) подразумевает, что вы должны сначала обрисовать ситуацию, затем объяснить свои действия и, наконец, описать результат.

Пример: «Расскажите мне о проблемных взаимодействиях с коллегами по команде».

Ситуация: При работе над операционной системой мне довелось работать вместе с тремя коллегами. Два человека были превосходны, а вот о третьем я такого сказать не могу. Он был замкнутым, редко участвовал в обсуждениях и с трудом справлялся со своей частью работы. Это создавало проблемы: во-первых, на нашу долю доставалось больше работы, а во-вторых, мы не знали, можно ли на него рассчитывать.

Действие: Я не хотел исключать его из команды, поэтому постарался решить проблему. Для этого я сделал три вещи.

Сначала нужно было понять, почему он так себя ведет. Лень? Нехватка времени? Я завязал с ним беседу и задал наводящие вопросы относительно того, что он думает о происходящем. Оказалось, что это вовсе не лень; просто ему не хватает уверенности в своих силах.

Поняв причину, я постарался объяснить ему, что бояться неудач не стоит. Я рассказал о некоторых своих ошибках и признался, что некоторые места текущего проекта мне тоже понятны не до конца.

Наконец, я попросил его помочь мне в работе над некоторыми компонентами проекта. Мы вместе разработали подробную спецификацию одного серьезного компонента — намного более подробную, чем до того. Когда все встало на свои места, он понял, что проект не так страшен, как казалось на первый взгляд.

Результат: После того как его уверенность укрепилась, этот человек стал чаще браться за менее ответственные части проекта, но со временем стал справляться и с более серьезными задачами. Он выполнял всю работу вовремя и принимал

участие в обсуждениях. Мы с радостью согласились работать с ним над следующим проектом.

Описания ситуации и результата должны быть очень краткими. Вашему интервьюеру не нужны лишние подробности — он в них только запутается.

При использовании модели SAR интервьюер легко понимает, какой была ситуация, каковы ваши действия и что получилось в результате.

Попробуйте свести свои истории в таблицу:

	Суть	Ситуация	Действия	Результат	Выводы
История 1			1. ... 2. ... 3. ...		
История 2					

Почти всегда «действие» оказывается самой важной частью истории. К сожалению, слишком многие кандидаты подолгу расписывают ситуацию, а потом едва упоминают о действиях.

Уделите особое внимание действиям. Там, где это возможно, разбейте их на несколько частей, например: «Тогда я сделал три вещи. Сначала я...» Это придаст объяснению глубины.

Итак, расскажите о себе...

Многие интервьюеры в начале собеседования предлагают немного рассказать о себе или пройтись по пунктам вашего резюме. Эта «затравка» определит первое впечатление о вас, поэтому будет важно правильно построить рассказ.

Многим людям хорошо подходит хронологическая структура: первое предложение описывает текущую работу, а в заключение упоминаются актуальные и интересные увлечения, не относящиеся к работе (если они есть).

1. **Текущая должность**: «Я работаю программистом в Microworks, где возглавлял группу Android-разработки последние пять лет».
2. **Образование**: «Я получил диплом в Беркли, проходил практику в нескольких небольших фирмах».
3. **Опыт работы**: «После окончания университета мне захотелось поработать в крупной корпорации, поэтому я поступил на работу в Amazon. Опыт оказался очень полезным: я много узнал о проектировании больших систем, и мне даже довелось управлять разработкой одного из ключевых компонентов AWS».
4. **Текущая должность (подробнее)**: «Один из моих бывших начальников из Amazon пригласил меня в свою фирму, из которой я и перешел в Microworks. Здесь я занимался исходной архитектурой системы, а затем воспользовался возможностью возглавить группу Android. У меня трое подчиненных, но в основном я занимаюсь чисто техническим руководством: архитектура, программирование и т. д.».

5. **В свободное время**: «В свободное время я изучаю программирование для iOS, а также модерирую некоторые форумы, посвященные программированию для Android».
6. **Итог**: «Сейчас я ищу что-нибудь новое, и ваша компания привлекла мое внимание. Мне всегда нравилась связь с пользователями, и мне хотелось бы вернуться в фирму поменьше».

Эта структура хорошо подходит для 95% кандидатов. Вероятно, кандидаты с большим послужным списком могут сократить некоторые пункты: «После получения диплома я проработал несколько лет в Amazon, а потом перешел в начинающую фирму, где возглавил группу Android».

VI

«O» большое

Эта концепция настолько важна, что в книге ей посвящена целая (и довольно длинная!) глава.

«O» большое — метрика, используемая для описания эффективности алгоритмов. Плохое понимание этой темы сильно вредит навыкам разработки алгоритмов. Мало того что оно произведет плохое впечатление на собеседовании, — такому разработчику будет трудно понять, в каких ситуациях его алгоритм начинает работать быстрее или медленнее.

Аналогия

Представьте следующий сценарий: вам нужно передать файл, хранящийся на жестком диске, вашему знакомому на другом конце страны. Файл должен быть передан как можно быстрее. Как отправить его?

Первое, что приходит в голову, — электронная почта, FTP или другой механизм передачи данных по сети. Мысль разумная, но правильная лишь отчасти.

Бесспорно, для небольших файлов этот способ годится. Чтобы добраться до аэропорта, сесть на самолет и доставить файл, потребуется 5–10 часов. Но что, если файл очень, очень большой? Может ли оказаться, что его «физическая» доставка самолетом получится более быстрой?

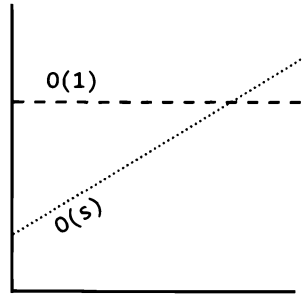
Да, может. Электронная передача терабайтного файла может занять больше суток. Вполне возможно, что перелет произойдет быстрее. Если файл исключительно важен (а затраты не учитываются), будет выбран этот вариант.

Временная сложность

Для описания «алгоритма» передачи данных можно воспользоваться понятием *асимптотической сложности*, или «записью *"O" большого*». Время «алгоритма» передачи данных можно описать следующим образом:

- **Передача по сети**: $O(s)$, где s — размер файла. Это означает, что время передачи файла возрастает линейно с размером файла. (Такая формулировка немного упрощена, но для наших целей подойдет.)
- **Пересылка самолетом**: $O(1)$ относительно размера файла. С увеличением размера файла время его доставки не увеличивается, а остается постоянным.

Какой бы большой ни была константа и как бы медленно ни происходил линейный рост, в какой-то момент он превысит константу.



Существует много других видов сложности. Среди наиболее распространенных можно выделить $O(\log N)$, $O(N \log N)$, $O(N)$, $O(N^2)$ и $O(2^N)$. Впрочем, какого-то фиксированного списка возможных вариантов не существует.

Сложность может включать несколько переменных. Например, время покраски забора шириной `w` метров и высотой `h` метров можно описать в виде $O(wh)$, а если забор должен быть покрыт краской в `p` слоев, можно сказать, что время составляет $O(whp)$.

O, Θ и Ω

В научной среде обозначения O, Θ и Ω применяются для описания сложности алгоритмов.

O: в научных трудах O описывает верхнюю границу вычислительной сложности. Сложность алгоритма, выводящего все значения из массива, может быть описана обозначением $O(N)$, но также ее можно описать обозначениями $O(N^2)$, $O(N^3)$ или $O(2^N)$ (и многими другими). Алгоритм по крайней мере не превосходит их по скорости; следовательно, все эти варианты описывают верхнюю границу сложности. Ситуацию можно сравнить с отношениями «меньше либо равно»: если Бобу исполнилось `X` лет (будем считать, что предельный срок жизни равен 130 годам), то можно утверждать, что `X ≤ 130`. Также будет правильно утверждать, что `X ≤ 1000` или `X ≤ 1 000 000`. Формально эти утверждения верны (хотя особой пользы и не приносят).

Ω: в научных трудах Ω представляет эквивалентную концепцию для нижней границы. Сложность вывода значений из массива может быть описана обозначением $\Omega(N)$, равно как и обозначениями $\Omega(\log N)$ и $\Omega(1)$. Вы уверены в том, что быстрее она работать не будет.

Θ: в научных трудах Θ подразумевает как O, так и Ω. Иначе говоря, алгоритм имеет сложность $\Theta(N)$, если он одновременно обладает сложностью $O(N)$ и $\Omega(N)$. Θ определяет точную границу сложности.

На практике (а следовательно, и на собеседованиях) смысл Θ и O практически смешался. Представления об O в среде разработчиков ближе к тому, что теоретики подразумевают под Θ, так что описание сложности вывода массива в виде $O(N^2)$ будет воспринято как ошибочное. Вероятно, специалист-практик скажет, что эта операция имеет сложность $O(N)$.

В этой книге запись «О» большого будет использоваться в том смысле, который принят среди практиков: как точное описание границ сложности.

Лучший, худший и ожидаемый случай

Время выполнения алгоритма, то есть его временную сложность, можно описать тремя разными способами.

Рассмотрим ситуацию на примере быстрой сортировки. Алгоритм быстрой сортировки выбирает случайный «опорный» элемент, а затем меняет местами значения в массиве таким образом, чтобы элементы, меньшие опорного элемента, предшествовали тем элементам, значения которых превышают значение опорного. Далее левая и правая части полученного «частично отсортированного» массива рекурсивно сортируются по той же схеме.

- **Лучший случай**: если все элементы равны, то алгоритм быстрой сортировки в среднем переберет массив всего один раз. Это соответствует времени $O(N)$. (На самом деле оценка частично зависит от реализации быстрой сортировки — существуют реализации, которые очень быстро работают с отсортированными массивами.)
- **Худший случай**: а если в результате хронического невезения в качестве опорного будет постоянно выбираться наибольший элемент массива? (Кстати, это вполне возможно, если в качестве опорного выбирается первый элемент подмассива, а сам массив отсортирован в обратном порядке.) Тогда рекурсия не делит массив надвое, а всего лишь уменьшает подмассив на один элемент. В этом вырожденном случае время выполнения достигает $O(N^2)$.
- **Ожидаемый случай**: впрочем, обычно такие замечательные или ужасные ситуации не встречаются. Конечно, в отдельных случаях опорное значение может оказаться очень высоким или низким, но такие аномалии не будут повторяться снова и снова. В среднем время выполнения составит $O(N \log N)$.

Сложность для лучшего случая в книге почти не обсуждается, потому что особой пользы эта концепция не принесет. В конце концов, можно взять практически любой алгоритм, передать ему специально подобранные данные и получить время $O(1)$ в лучшем случае. Для многих (а возможно, и для большинства) алгоритмов худший случай совпадает с ожидаемым. Иногда они отличаются, и тогда приходится давать обе оценки.

Какими отношениями связаны лучший/худший/ожидаемый случай и $O/\Theta/\Omega$?

Кандидаты часто путают эти концепции (вероятно, потому, что в них присутствуют похожие понятия «выше»/«ниже»/«ровно»), но никакой конкретной связи между ними нет. Лучший, худший и ожидаемый случаи описывают сложность O (или Θ) для конкретных входных данных или сценариев.

O, Ω и Θ описывают верхнюю, нижнюю и точную границу сложности.

Пространственная сложность

Время не единственный показатель эффективности алгоритма. Также может быть важен объем памяти (пространство), необходимый алгоритму.

Пространственная сложность — концепция, параллельная временной сложности. Если вам потребуется создать массив размера `n`, это потребует памяти $O(n)$. Для создания массива размера `n×n` потребуется память $O(n^2)$.

Также учитываются затраты памяти в стеке. Например, следующий код потребует времени $O(n)$ и памяти $O(n)$.

```
1  int sum(int n) { /* Пример 1.*/
2    if (n <= 0) {
3      return 0;
4    }
5    return n + sum(n-1);
6  }
```

Каждый вызов добавляет новый уровень в стек.

```
1  sum(4)
2    -> sum(3)
3      -> sum(2)
4      -> sum(1)
5          -> sum(0)
```

Каждый вызов помещается в стек и занимает память.

Тем не менее наличие n вызовов еще не подразумевает затрат памяти $O(n)$. Рассмотрим следующую функцию для суммирования соседних элементов в интервале от 0 до n:

```
1  int pairSumSequence(int n) { /* Пример 2.*/
2    int sum = 0;
3    for (int i = 0; i < n; i++) {
4      sum += pairSum(i, i + 1);
5    }
6    return sum;
7  }
8
9  int pairSum(int a, int b) {
10   return a + b;
11 }
```

Всего будет сделано примерно $O(n)$ вызовов `pairSum`. Однако эти вызовы не размещаются в стеке одновременно, поэтому затраты памяти составят только $O(1)$.

Константы

Код со сложностью $O(N)$ для конкретного ввода вполне может работать быстрее кода $O(1)$. Синтаксис «O» большого описывает только скорость роста. По этой причине константы исключаются из описания сложности. Запись вида $O(2N)$ сокращается до $O(N)$.

Многие разработчики никак не привыкнут к этому. Они видят, что код содержит два (не вложенных) цикла for, и пишут $O(2N)$. Им кажется, что такая запись «точнее». На самом деле это не так.

Возьмем следующий пример:

```
(1)

1  int min = Integer.MAX_VALUE;
2  int max = Integer.MIN_VALUE;
3  for (int x : array) {
4    if (x < min) min = x;
```

```
5    if (x > max) max = x;
6  }
```

(2)

```
1  int min = Integer.MAX_VALUE;
2  int max = Integer.MIN_VALUE;
3  for (int x : array) {
4    if (x < min) min = x;
5  }
6  for (int x : array) {
7    if (x > max) max = x;
8  }
```

Какой вариант работает быстрее? Первый содержит один цикл `for`, второй — два цикла `for`. Но при этом в первом варианте цикл `for` состоит из двух строк вместо одной. Но если заниматься подсчетом команд, придется переходить на уровень ассемблера и учитывать то, что умножение требует большего количества команд, чем сложение, учитывать оптимизации компилятора и множество других подробностей.

Все это невероятно усложняется, так что даже не вставайте на этот путь. Синтаксис «О» большого выражает закономерность масштабирования сложности. Просто следует понимать, что $O(N)$ не всегда лучше $O(N^2)$.

Исключение второстепенных факторов

Что делать с выражениями вида $O(N^2 + N)$? Второе N константой не является, однако оно не играет определяющей роли.

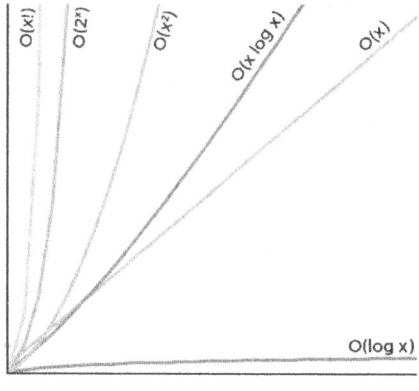
Ранее уже говорилось о том, что константы исключаются из формулы. Следовательно, запись $O(N^2 + N^2)$ превращается в $O(N^2)$? Если мы исключаем второе слагаемое N^2, то зачем оставлять N?

Второстепенные (не доминирующие) факторы исключаются из выражения:

- $O(N^2 + N)$ превращается в $O(N^2)$.
- $O(N + \log N)$ превращается в $O(N)$.
- $O(5*2^N + 1000N^{100})$ превращается в $O(2^N)$.

Это не означает, что в выражении сложности не может быть суммы. Например, выражение $O(B^2 + A)$ сократить не удастся (без дополнительной информации об A и B).

На следующем графике изображена скорость роста некоторых типичных вариантов сложности.



Как видно из графика, $O(x^2)$ растет намного быстрее $O(x)$, но даже отдаленно не так быстро, как $O(2^x)$ или $O(x!)$. Существует много других вариантов сложности, худших $O(x!)$, например $O(x^x)$ или $O(2^x * x!)$.

Составные алгоритмы: сложение и умножение

Допустим, алгоритм состоит из двух шагов. В каких случаях сложности следует умножать, а в каких случаях они суммируются? Это еще один источник частых ошибок на собеседованиях.

Суммирование сложности: O(A + B)

```
1  for (int a : arrA) {
2    print(a);
3  }
4
5  for (int b : arrB) {
6    print(b);
7  }
```

Умножение сложности: O(A*B)

```
1  for (int a : arrA) {
2    for (int b : arrB) {
3      print(a + "," + b);
4    }
5  }
```

В первом примере сначала выполняются `A` блоков работы, а затем `B` блоков работы. Следовательно, общий объем работы составит $O(A + B)$.

Во втором примере для каждого элемента `A` выполняются `B` блоков работы. Следовательно, общий объем работы составит $O(A * B)$.

Другими словами:

- Если алгоритм построен по принципу «делаем это, а когда работа будет закончена, делаем то», сложности суммируются.
- Если алгоритм построен по принципу «каждый раз, когда делается это, сделать то», сложности перемножаются.

На собеседовании очень легко запутаться в этой теме, так что будьте осторожны.

Амортизированное время

`ArrayList`, массив с динамически изменяемым размером, позволяет использовать преимущества массива с сохранением гибкости в выборе размера. При работе с `ArrayList` свободное место не кончится, потому что емкость будет увеличиваться по мере вставки элементов.

Класс `ArrayList` реализуется на базе массива. Когда массив заполняется, класс `ArrayList` создает новый массив двойной емкости и копирует все элементы в новый массив. Как оценить время выполнения вставки? Это непростой вопрос.

Массив может быть заполнен. Если массив содержит `N` элементов, то вставка нового элемента будет выполняться за время $O(N)$: необходимо создать новый массив размера `2N` и скопировать в него `N` элементов. Вставка займет время $O(N)$.

Однако мы знаем, что заполнение массива происходит относительно редко. В подавляющем большинстве случаев вставка будет выполняться за время $O(1)$.

Концепция *амортизированного времени* учитывает обе возможности. Амортизированное время позволяет указать, что худший случай происходит время от времени. Но после того как он произошел, до его повторного возникновения пройдет так много времени, что лишние затраты «амортизируются».

Каким будет амортизированное время в данном случае?

При вставке элементов емкость удваивается, когда размер массива является степенью 2. Таким образом, после `X` элементов емкость массива удваивается при размере 1, 2, 4, 8, 16, ..., `X`. Удвоение требует соответственно 1, 2, 4, 8, 16, 32, 64, ..., `X` копий.

Чему равна сумма 1 + 2 + 4 + 8 + 16 + ... + `X`? Если прочитать ее слева направо, последовательность начинается с 1 и удваивается, пока не достигнет `X`. Если читать справа налево, последовательность начинается с `X` и уменьшается вдвое, пока не достигнет 1. Чему равна сумма `X + X/2 + X/4 + X/8 + . . . + 1`? Приблизительно `2X`.

Таким образом, `X` вставок выполняются за время $O(2X)$. Амортизированное время каждой вставки равно $O(1)$.

Сложность Log N

В оценке сложности часто встречается выражение $O(\log N)$. Откуда оно берется?

Для примера возьмем бинарный поиск. При бинарном поиске значение `x` ищется в отсортированном массиве из `N` элементов. Сначала `x` сравнивается с серединой массива. Если `x` и средний элемент равны, поиск завершается. Если `x` меньше среднего элемента, то поиск проводится в левой части массива, а если больше — то в правой.

```
искать 9 в {1, 5, 8, 9, 11, 13, 15, 19, 21}
  сравнить 9 с 11 -> меньше
  искать 9 в {1, 5, 8, 9, 11}
    сравнить 9 с 8 -> больше
    искать 9 в {9, 11}
      сравнить 9 с 9
    вернуть результат
```

Сначала поиск ведется в массиве с `N` элементами. После одного шага количество элементов уменьшается до `N/2`. Еще через один шаг остается `N/4` элементов и т. д. Мы останавливаемся либо когда значение будет найдено, либо когда останется всего один элемент.

Таким образом, общее время выполнения определяется количеством шагов (последовательных делений `N` на 2), после которых `N` уменьшится до 1.

```
N = 16
N = 8    /* разделить на 2 */
N = 4    /* разделить на */
N = 2    /* разделить на 2 */
N = 1    /* разделить на 2 */
```

Также можно рассматривать происходящее в обратном направлении (переход от 1 к 16). Сколько раз нужно умножить 1 на 2, чтобы получить `N`?

```
N = 1
N = 2    /* умножить на 2 */
N = 4    /* умножить на 2 */
N = 8    /* умножить на 2 */
N = 16   /* умножить на 2 */
```

Чему равно k в выражении $2^k = N$? По определению логарифма

$2^4 = 16 \rightarrow \log_2 16 = 4$
$\log_2 N = k \rightarrow 2^k = N$

Полезный практический совет: когда вы встречаете задачу, в которой количество элементов последовательно делится надвое, скорее всего, время выполнения составит $O(\log N)$.

По этой же причине поиск элемента в сбалансированном дереве бинарного поиска выполняется за время $O(\log N)$. После каждого сравнения происходит переход налево или направо. С каждой стороны находится половина узлов, поэтому пространство задачи каждый раз сокращается вдвое.

По какому основанию вычисляется логарифм? Отличный вопрос. В двух словах: для записи «O» большого это не имеет значения. Более подробное объяснение приводится в разделе «Основания логарифмов» на с. 667.

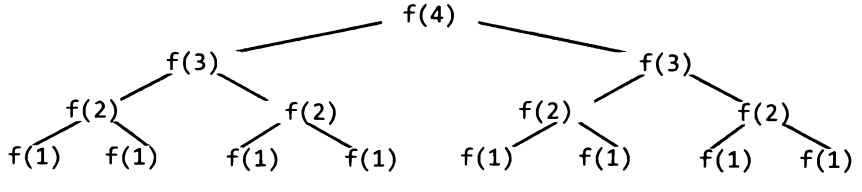
Сложность рекурсивных алгоритмов

А вот более сложный вопрос: как вы оцените время выполнения следующего кода?

```
1  int f(int n) {
2    if (n <= 0) {
3      return 1;
4    }
5    return f(n - 1) + f(n - 1);
6  }
```

Многие кандидаты видят два вызова f и почему-то предлагают результат $O(N^2)$. Это полностью неверно.

Вместо того чтобы гадать, вычислим время выполнения на основании анализа кода. Допустим, вы вызвали f(4). Этот вызов порождает два вызова f(3). Каждый из вызовов f(3) вызывает f(2), пока мы не доберемся до f(1).



Сколько узлов (вызовов) в этом дереве? Дерево имеет глубину N. У каждого узла (то есть вызова функции) два потомка. Следовательно, на каждом уровне количество узлов вдвое больше количества узлов на предыдущем уровне. Количество узлов на каждом уровне равно:

Уровень	Количество узлов	Также выражается в виде...	Или...
0	1		2^0
1	2	2 * предыдущий уровень = 2	2^1
2	4	2 * предыдущий уровень = $2 * 2^1 = 2^2$	2^2
3	8	2 * предыдущий уровень = $2 * 2^2 = 2^3$	2^3
4	16	2 * предыдущий уровень = $2 * 2^3 = 2^4$	2^4

Следовательно, всего будет $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^N$ узлов, что равно $2^{N+1} - 1$. (См. «Сумма степеней 2», с. 666.)

Постарайтесь запомнить эту закономерность. Если вы используете рекурсивную функцию, которая порождает несколько вызовов, время выполнения часто (хотя и не всегда) имеет вид $O(ветви^{глубина})$, где *ветви* — количество ветвлений при каждом рекурсивном вызове. В нашем примере получается $O(2^N)$.

Как было сказано ранее, основание логарифма для «О» не имеет большого значения, так как логарифмы по разным основаниям различаются только постоянным множителем. С другой стороны, к экспоненте это не относится; основание экспоненты имеет значение. Сравните 2^n и 8^n. Раскрыв 8^n, получаем $(2^3)^n$, то есть 2^{3n}, или $2^{2n} * 2^n$. Как видите, 8^n и 2^n отличаются множителем 2^{2n} — ничего общего с постоянным множителем!

Пространственная сложность этого алгоритма составит $O(N)$. Хотя дерево в сумме содержит $O(2^N)$ узлов, в любой момент существуют только $O(N)$ узлов. Следовательно, затраты памяти ограничиваются величиной $O(N)$.

VII

Технические вопросы

Технические вопросы образуют основу собеседования во многих компаниях. Часто кандидатов страшат сложные задачи, однако и к таким задачам можно подойти логически.

Как организовать подготовку

Многие кандидаты считают, что достаточно просмотреть задачи и готовые решения. Это напоминает попытку освоить математический анализ, читая задачи и ответы к ним. От запоминания решений толку все равно не будет.

Для каждой задачи из этой книги (и любой другой задачи, которая вам попадется):

- *Попытайтесь решить задачу самостоятельно.* В конце книги приведены подсказки, но вы должны пересилить соблазн и постараться разработать решение без посторонней помощи. Вам встретится множество сложных заданий — и это нормально! Решая задачу, помните о затратах времени и памяти.
- *Запишите код алгоритма на бумаге.* Вы всю жизнь программировали на компьютере, и это, безусловно, хорошо. Но на собеседовании вам не поможет ни выделение синтаксиса, ни автозавершение, ни быстрая отладка. Привыкайте к этому — и к тому, как медленно пишется код вручную.
- *Протестируйте свой код — на бумаге.* Протестируйте общие случаи, лучшие случаи, базовые случаи, ошибки и т. д. Вам придется это делать на собеседовании, поэтому лучше подготовиться заранее.
- *Введите написанный код в компьютер «как есть».* Вероятно, вы обнаружите множество ошибок. Проанализируйте все ошибки и сделайте все, чтобы на настоящем собеседовании их не допустить.

Очень полезны имитации собеседований. Пригласите своих друзей и поочередно проведите «собеседования» друг с другом. Хотя ваши друзья не являются профессиональными интервьюерами, они в состоянии проверить решения задач на программирование и алгоритмизацию. Кроме того, вы узнаете много нового, побывав на месте интервьюера.

Что нужно знать

Большинство интервьюеров не будут задавать вопросы о конкретных алгоритмах балансировки двоичного дерева или других сложных алгоритмах. Честно говоря, они сами их уже забыли, ведь такие вещи забываются сразу после окончания учебы.

Вам нужно знать основы. Ниже перечислены абсолютно необходимые, обязательные знания.

Структуры данных	Алгоритмы	Концепции
Связные списки	Поиск в ширину	Операции с битами
Деревья, нагруженные деревья, графы	Поиск в глубину	Память (стек, куча)
Стеки и очереди	Бинарный поиск	Рекурсия
Кучи	Сортировка слиянием	Динамическое программирование
Векторы/ArrayList	Быстрая сортировка	Временная и пространственная сложность (запись «О»-большое»)

Убедитесь, что вы понимаете, как использовать и реализовывать все эти алгоритмы, а там, где это существенно, — знаете их временную и пространственную сложность.

Потренируйтесь в реализациях структур данных и алгоритмов (на бумаге, потом на компьютере), это очень полезное упражнение. Оно поможет вам лучше понять внутреннюю реализацию многих структур данных, а это очень важно на собеседованиях.

А вы не пропустили предыдущий абзац? Он важен. Если вы не чувствуете себя очень, очень уверенно со всеми перечисленными структурами данных и алгоритмами, потренируйтесь в их создании с нуля.

Особенно важны хеш-таблицы. Убедитесь в том, что знаете эту структуру данных, как свои пять пальцев.

Таблица степеней двойки

Следующая таблица пригодится во многих задачах, связанных с масштабируемостью или ограничениями памяти. Заучивать ее не обязательно, хотя и это может быть полезно. По крайней мере, вы должны уверенно вывести нужное значение.

Степень 2	Точное значение (X)	Приближенное значение	X байтов в мегабайте, гигабайте и т. д.
7	128		
8	256		
10	1024	1 тысяча	1 Кбайт
16	65 536		64 Кбайт
20	1 048 536	1 миллион	1 Мбайт
30	1 073 741 824	1 миллиард	1 Гбайт
32	4 294 967 296		4 Гбайт
40	1 099 511 627 776	1 триллион	1 Тбайт

Например, по этой таблице можно легко рассчитать, хватит ли имеющегося объема памяти для хеш-таблицы, отображающей каждое 32-битное целое число на логическое значение. Всего существует 2^{32} таких чисел. Так как каждое число занимает

один бит, для хранения данных потребуется 2^{32} бит (или 2^{29} байт). Это соответствует приблизительно половине гигабайта памяти, поэтому на типичной машине данные легко поместятся в памяти.

Если вы проходите телефонное собеседование в веб-ориентированной компании, полезно держать эту таблицу перед глазами.

Процесс решения задачи

На следующей схеме показано, как следует подходить к решению задач. Используйте ее в своей практике (схему и другие дополнительные материалы можно загрузить на сайте *CrackingTheCodingInterview.com*).

Разберем эту схему более подробно.

Чего ожидать

Собеседование не может быть простым. Если вам не удается с ходу дать ответ на все вопросы (и даже на один вопрос!) — это нормально!

Внимательно слушайте интервьюера. Он может играть более или менее активную роль в решении задач. Уровень его участия зависит от вашей результативности, сложности вопроса, того, что интересует интервьюера, и его личных качеств.

Когда вы беретесь за задачу на собеседовании (или тренируетесь в решении), действуйте методично по следующей схеме.

1. Внимательно слушайте

Вероятно, вы уже слышали этот совет прежде, но я имею в виду нечто большее, чем стандартный совет «убедитесь в том, что вы правильно расслышали формулировку задачи».

Да, задачу нужно выслушать и убедиться в том, что вы расслышали все правильно. И нужно задать вопросы по всем темам, в которых вы не уверены.

Но я имею в виду нечто большее.

Внимательно слушайте задачу и мысленно отметьте всю уникальную информацию в задаче.

Предположим, вопрос начинается с одной из следующих строк. Разумно предположить, что информация приведена не просто так:

- «Имеются два отсортированных массива. Требуется найти...»

  Вероятно, тот факт, что данные отсортированы, должен использоваться в задаче. Скорее всего, оптимальный алгоритм для отсортированных данных отличается от оптимального алгоритма для ситуации без сортировки.

- «Разработайте алгоритм для многократного выполнения на сервере...»

  Ситуация с многократным выполнением алгоритма на сервере отличается от однократного запуска. Возможно, данные стоит кэшировать? А может, в такой ситуации будет оправданна разумная предварительная обработка исходного набора данных?

Схема решения задачи

1 СЛУШАЙТЕ

Очень внимательно изучите всю информацию, содержащуюся в описании задачи. Вероятно, вся эта информация понадобится для создания оптимального алгоритма.

Оптимизация BUD
«Узкие места» (Bottlenecks)
Лишняя работа (Unnecessary work)
Повторяющаяся работа (Duplicated work)

2 ПРЕДСТАВЬТЕ ПРИМЕР

Многие примеры слишком малы или отражают особые случаи. **Подвергните свой пример критическому анализу**. Не является ли он особым случаем в каком-либо отношении? Достаточно ли он велик?

3 ОПИШИТЕ РЕШЕНИЕ МЕТОДОМ «ГРУБОЙ СИЛЫ»

Постарайтесь как можно раньше прийти к решению методом «грубой силы». Пока не беспокойтесь о разработке эффективного алгоритма. Сначала опишите наивный алгоритм и его время выполнения, затем начинайте оптимизировать. Пока не пишите код!

4 ОПТИМИЗИРУЙТЕ

Примените к своему решению методом «грубой силы» оптимизацию BUD или попробуйте воспользоваться следующими рекомендациями:

- ❑ Поищите неиспользуемую информацию. Обычно в постановке задачи не бывает лишней информации.
- ❑ Решите задачу вручную для примера, а затем попробуйте восстановить свой мыслительный процесс. Как вы решали задачу?
- ❑ Решите задачу «неправильно», а потом подумайте, почему алгоритм не сработал. Можно ли устранить недостатки?
- ❑ Попробуйте найти компромисс между затратами времени и памяти. Хеш-таблицы в этом особенно полезны!

5 ПРОВЕДИТЕ ПОШАГОВЫЙ РАЗБОР

Теперь, когда у вас появилось оптимальное решение, **подробно изучите каждый его шаг**. Убедитесь в том, что вы понимаете каждую мелочь, прежде чем браться за программирование.

6 НАПИШИТЕ КОД

Ваша цель — **написать красивый код**. Применяйте модульный подход с самого начала и проведите рефакторинг, чтобы убрать все некрасивое.

Не молчите! Интервьюер хочет слышать, как вы подходите к решению задачи.

7 ПРОТЕСТИРУЙТЕ СВОЕ РЕШЕНИЕ

Тестирование ведется в следующем порядке:

1. Концептуальное тестирование. Пройдитесь по всему коду, как при подробном рецензировании кода.
2. Нетипичный или нестандартный код.
3. Проблемные места (арифметика, null-узлы).
4. Маленькие тестовые сценарии: они выполняются намного быстрее больших тестовых сценариев, не уступая им в эффективности.
5. Особые и граничные случаи.

А когда найдете ошибки — **исправляйте аккуратно**!

Вряд ли интервьюер будет давать информацию, не влияющую на алгоритм (хотя это и не исключено).

Многие кандидаты слышат все правильно. Но после 10 минут, проведенных за разработкой алгоритма, некоторые ключевые подробности могут быть забыты. И тогда может оказаться, что найти оптимальное решение уже не удастся.

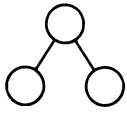
Ваш первый алгоритм не обязан использовать всю информацию. Но если вы оказались в тупике или пытаетесь найти более эффективное решение, спросите себя, использовали ли вы всю информацию в задаче. Возможно, важнейшую информацию даже стоит записать на доске.

2. Представьте пример

Пример может серьезно помочь в поиске ответа, и все же многие кандидаты пытаются решать задачи «в уме».

Когда вы услышали вопрос, встаньте с кресла, подойдите к доске и нарисуйте пример. Впрочем, это не так просто — пример должен быть хорошим.

Очень часто кандидат, которому предложили нарисовать бинарное дерево поиска, рисует нечто такое:



Этот пример плох по нескольким причинам. Во-первых, он слишком мал; в таком примере трудно найти закономерность. Во-вторых, он не конкретен. Бинарное дерево поиска содержит значения. Что, если числа сообщат что-то полезное о том, как решать задачу? В-третьих, нарисованное дерево является особым случаем: мало того, что оно сбалансировано; это идеальное дерево, в котором каждый узел, кроме листовых, имеет два дочерних узла. Особые случаи часто вводят в заблуждение.

Вместо этого ваш пример должен быть:

- **Конкретным**. В нем должны использоваться реальные числа или строки (если это относится к вашей задаче).
- **Достаточно большим**. Многие примеры слишком малы (вдвое меньше необходимого).
- **Не относящимся к особым случаям**. Будьте осторожны: очень легко непреднамеренно нарисовать особый случай. Если ваш пример в каком-то отношении может рассматриваться как частный случай (даже если вам кажется, что это ни на что не повлияет), исправьте этот недостаток.

Постарайтесь создать самый лучший пример, который у вас получится. Если позднее окажется, что ваш пример недостаточно хорош, обязательно исправьте его.

3. Опишите решение методом «грубой силы»

Разобравшись с примером (вообще говоря, в некоторых задачах порядок шагов 2 и 3 может быть обратным), опишите решение методом «грубой силы». Интервьюер ожидает, что исходный алгоритм не будет оптимальным, — это нормально.

Некоторые кандидаты не предлагают решение методом «грубой силы», потому то считают его очевидным и очень плохим. Но то, что очевидно для вас, может быть не очевидно для других. Интервьюер не должен думать, что вы не можете найти даже простое решение.

Ничего страшного, что начальное решение получилось плохим. Опишите его пространственную и временную сложность, а потом переходите к усовершенствованию.

Алгоритм «грубой силы», несмотря на возможную низкую эффективность, заслуживает обсуждения. Он является отправной точкой для оптимизации и помогает лучше понять суть проблемы.

4. Переходите к оптимизации

Когда решение методом «грубой силы» будет готово, переходите к оптимизации. Несколько приемов, которые часто оказываются эффективными:

1. Поищите неиспользуемую информацию. Упоминал ли интервьюер о том, что массив отсортирован? Как использовать этот факт в решении?
2. Найдите новый пример. Иногда даже рассмотрение другого примера помогает выйти из тупика и обнаружить закономерность в задаче.
3. Решите задачу «неправильно». Подобно тому как неэффективное решение помогает в поиске эффективного, неправильное решение может помочь в поиске правильного. Например, если вам предложено сгенерировать случайный элемент из набора равновероятных значений, неправильное решение может возвращать псевдослучайный результат: возвращаться может любое значение, но некоторые значения более вероятны, чем другие. Тогда можно задуматься над тем, почему это решение не является полностью случайным. Можно ли сбалансировать вероятности?
4. Поищите компромисс между затратами времени и памяти. Иногда хранение дополнительной информации состояния в задаче помогает оптимизировать время выполнения.
5. Проведите предварительную обработку. Возможно ли реорганизовать данные (отсортировать и т. д.) или вычислить некоторые значения заранее, чтобы сэкономить время в долгосрочной перспективе?
6. Воспользуйтесь хеш-таблицей. Хеш-таблицы часто встречаются в вопросах на собеседованиях и должны первыми приходить вам на ум.
7. Подумайте над лучшим ожидаемым временем.

Проанализируйте решение методом «грубой силы» с учетом этих рекомендаций. Попробуйте исправить недостатки BUD (с. 117).

5. Проведите пошаговый разбор

Когда оптимальный алгоритм будет найден, не торопитесь переходить к написанию кода. Выделите немного времени на то, чтобы укрепить свое понимание алгоритма.

Программирование «у доски» проходит медленно, очень медленно. Как и тестирование кода и исправление ошибок. Как следствие, вы должны позаботиться о том, чтобы решение было с самого начала как можно ближе к идеалу.

Пройдитесь по всему алгоритму и составьте представление о структуре кода. Вы должны знать все переменные и понимать, когда они изменяются.

При желании напишите псевдокод. Будьте внимательны со стилем. Основные шаги («(1) Провести поиск в массиве. (2) Найти максимальный элемент. (3) Вставить в кучу») или сокращенная логика («if p < q, переместить p иначе переместить q») могут быть полезны. Но если в вашем псевдокоде появляются циклы `for`, записанные на английском языке, толку от него будет немного. Вероятно, быстрее просто написать программный код.

6. Напишите код

Итак, у вас есть оптимальный алгоритм и вы точно знаете, что собираетесь написать. Можно браться за дело.

Начинайте с левого верхнего угла доски (вам понадобится место). Избегайте «сползающих строк» — такой код выглядит неряшливо и может вызвать путаницу в языках, чувствительных к пробелам (например, в Python).

Помните, что короткий фрагмент кода должен продемонстрировать, что вы являетесь выдающимся разработчиком. Учитываются все аспекты — ваш код должен быть красивым.

- **Модульность** — признак хорошего стиля программирования. Кроме того, она упрощает вашу задачу. Если в вашем алгоритме используется матрица, инициализированная элементами `{{1, 2, 3}, {4, 5, 6}, ...}`, не тратьте время на написание кода инициализации. Просто считайте, что у вас есть функция `initIncrementalMatrix(int size)`. Подробности можно будет заполнить позднее, если понадобится.
- **Проверка ошибок** — одни интервьюеры обращают на проверку ошибок большое внимание, другие нет. Хороший компромиссный способ — добавить комментарии `TODO` и вслух объяснить, что вы собираетесь проверять.
- **Использование других классов/структур**, где это уместно. Если функция должна вернуть список начальных и конечных точек, можно воспользоваться двумерным массивом. Тем не менее лучше передать данные в виде списка объектов `StartEndPair` (или `Range`). Приводить подробное описание для каждого класса не обязательно. Просто считайте, что класс существует, и займитесь подробностями позднее, если останется время.
- **Хорошие имена переменных.** Код, в котором везде используются однобуквенные переменные, плохо читается. Это не означает, что вы не должны использовать переменные `i` и `j` там, где это уместно (например, в базовом цикле `for` для перебора элементов массива). Тем не менее будьте внимательны: в командах вида `i = startOfChild(array)` для переменной можно найти более удачное имя (например, `startChild`).

Конкретные представления о хорошем коде зависят от интервьюеров и кандидатов, а также от самой задачи. Сосредоточьтесь на написании красивого кода, что бы это ни означало для вас.

Если вы видите возможность последующего рефакторинга, упомяните о ней своему интервьюеру и решите, стоит ли тратить на это время. Обычно рефакторинг оправдан, но не всегда.

Если вы запутаетесь (что бывает довольно часто), вернитесь к своему примеру и снова разберите его.

7. Переходите к тестированию

Никто не станет публиковать код в реальном мире без тестирования. Точно так же нельзя «публиковать» код в ходе собеседования, не протестировав его. Впрочем, тестирование тоже бывает разным.

Многие кандидаты просто берут свой предыдущий пример и проверяют на нем свой код. Иногда такое тестирование выявляет ошибки, но оно требует много времени. Ручное тестирование происходит слишком медленно. Если для алгоритма берется действительно большой и качественный пример, обнаружение маленькой ошибки смещения на 1 в конце кода обойдется слишком дорого.

Вместо этого лучше применить следующий метод:

1. Начните с «концептуального» теста. Иначе говоря, просто прочитайте и проанализируйте каждую строку кода. Подумайте, как бы вы объяснили смысл кода рецензенту. Делает ли этот код то, что должен?
2. Займитесь поисками странно выглядящего кода. Лишний раз проверьте строку вида `x = length - 2`. Уделите особое внимание циклу `for`, который начинается с `i = 1`. Вероятно, это было сделано не просто так, но такие странности повышают риск коварных ошибок.
3. На основании своего опыта программирования вы уже знаете, где могут возникнуть проблемы: базовые случаи в рекурсивном коде; целочисленное деление; `null`-узлы в бинарных деревьях; начало и конец перебора связного списка... Лишний раз проверьте эти аспекты.
4. Маленькие тестовые сценарии. Здесь мы впервые применяем конкретный, реальный тестовый сценарий для тестирования кода. Не используйте большой 8-элементный массив, если можно обойтись массивом из 3 или 4 элементов. Вероятно, он обнаружит те же ошибки, но сделает это намного быстрее.
5. Особые случаи. Протестируйте свой код на значения `null` или одноэлементные значения, граничные значения и другие особые случаи.

Конечно, если вы обнаружили ошибки (а это, скорее всего, неизбежно), их необходимо исправить. Не бросайтесь вносить первое исправление, которое вам придет в голову. Вместо этого тщательно проанализируйте, почему произошла ошибка, и убедитесь в том, что ваше решение является лучшим из всех.

Метод оптимизации 1: поиск BUD

Вероятно, это самый полезный метод оптимизации из всех известных мне. Под сокращением «BUD» скрываются «узкие места» (Bottlenecks), лишняя работа (Unnecessary work) и повторяющаяся работа (Duplicated work). Это три главные причины для неэффективного расходования времени в алгоритмах. Попробуйте проанализировать свое решение методом «грубой силы» и найти в нем эти аспекты. Обнаружив один из них, сосредоточьтесь на его устранении.

Если алгоритм по-прежнему не оптимален, примените этот метод к текущему лучшему алгоритму.

«Узкие места»

«Узким местом» называется часть алгоритма, которая вызывает снижение общего времени выполнения. Два наиболее распространенных случая:

- Существует однократно выполняемая работа, которая замедляет ваш алгоритм. Допустим, вы используете двухфазный алгоритм, который сначала сортирует массив, а потом выполняет поиск по некоторому свойству. Первая фаза выполняется за время $O(N \log N)$, а вторая — за время $O(N)$. Возможно, вторую фазу удастся сократить до $O(\log N)$ или даже $O(1)$, но отразится ли это на общем времени? Не особенно. Это второстепенный фактор, так как «узким местом» в данном случае является время $O(N \log N)$. До тех пор пока первая фаза не будет оптимизирована, общая сложность алгоритма так и останется равной $O(N \log N)$.
- Некоторая часть работы (например, поиск) выполняется многократно. Допустим, вам удастся сократить время $O(N)$ до $O(\log N)$ или даже $O(1)$. Такая оптимизация сильно сократит общее время выполнения.

Лишняя работа

Пример: вывести все положительные целочисленные решения уравнения $a^3 + b^3 = c^3 + d^3$, где a, b, c и d — целые числа в диапазоне от 1 до 1000.

Решение методом «грубой силы» просто использует четыре вложенных цикла:

```
1  n = 1000
2  for a from 1 to n
3    for b from 1 to n
4      for c from 1 to n
5        for d from 1 to n
6          if a³ + b³ == c³ + d³
7            print a, b, c, d
```

Алгоритм перебирает все возможные значения a, b, c и d и проверяет, работает ли текущая комбинация.

Продолжать проверку для других значений d не нужно — работает только одно. По крайней мере следует прервать цикл после нахождения действительного решения.

```
1  n = 1000
2  for a from 1 to n
3    for b from 1 to n
4      for c from 1 to n
5        for d from 1 to n
6          if a³ + b³ == c³ + d³
7            print a, b, c, d
8            break // Выход из цикла по d
```

Значительных изменений во времени выполнения ждать не стоит — алгоритм по-прежнему выполняется за время $O(N^4)$, но это хорошее и быстрое усовершенствование.

Осталась ли еще какая-нибудь лишняя работа? Да. Если для каждой комбинации (a, b, c) существует только одно возможное значение d, его можно просто вычислить по простой формуле:

$$d = \sqrt[3]{a^3 + b^3 - c^3}$$

```
1  n = 1000
2  for a from 1 to n
3    for b from 1 to n
4      for c from 1 to n
5        d = pow(a³ + b³ - c³, 1/3) // Округляется до int
6        if a³ + b³ == c³ + d³      // Проверить вычисленное значение
7          print a, b, c, d
```

Время выполнения сокращается с $O(N^4)$ до $O(N^3)$.

Повторяющаяся работа

Для рассмотрения повторяющейся работы мы воспользуемся той же задачей и алгоритмом «грубой силы».

По сути, алгоритм перебирает все пары (a, b), а затем ищет среди пар (c, d) совпадения для текущей пары (a, b).

Зачем продолжать вычислять все пары (c, d) для каждой пары (a, b)? Список пар (c, d) достаточно построить всего один раз. Тогда для каждой пары (a, b) поиск будет вестись по списку (c, d). Чтобы ускорить поиск совпадения, мы вставим каждую пару (c, d) в хеш-таблицу, связывающую сумму с парой (а вернее, со списком пар с данной суммой).

```
1  n = 1000
2  for c from 1 to n
3    for d from 1 to n
4      result = c³ + d³
5      append (c, d) to list at value map[result]
6  for a from 1 to n
7    for b from 1 to n
8      result = a³ + b³
9      list = map.get(result)
10      for each pair in list
11        print a, b, pair
```

Но если мы построили хеш со всеми парами (c, d), генерировать пары (a, b) не нужно — каждая пара (a, b) уже присутствует в хеше.

```
1  n = 1000
2  for c from 1 to n
3    for d from 1 to n
4      result = c³ + d³
5      append (c, d) to list at value map[result]
6
7  for each result, list in map
8    for each pair1 in list
9      for each pair2 in list
10       print pair1, pair2
```

В этом случае время выполнения сокращается до $O(N^2)$.

Метод оптимизации 2: интуитивный подход

Когда вы впервые (до знакомства с бинарным поиском) услышали о задаче поиска элемента в отсортированном массиве, вам вряд ли пришло в голову, что целевой массив нужно сравнить с серединой, а затем рекурсивно обработать подходящую половину.

Но если выдать человеку, не разбирающемуся в теории алгоритмов, упорядоченную по алфавиту стопку студенческих рефератов, он с большой вероятностью реализует некий аналог бинарного поиска. «Питер Смит? Пожалуй, это где-то в нижней части стопки». Он выберет случайный реферат в середине (или неподалеку от нее), сравнит имя с искомым, а затем продолжит действовать так с оставшейся частью рефератов. Ничего не зная о бинарном поиске, он интуитивно понимает его суть.

Наш мозг странно устроен. Стоит произнести фразу «Спроектируйте алгоритм», и многие люди впадают в ступор. Но если дать реальный пример — данные (например, массив) или их аналог из реального мира (стопка рефератов), как интуиция немедленно выдает вполне достойный алгоритм.

Я сталкивалась с этим явлением у многих кандидатов. Их компьютерные алгоритмы работают медленно, но стоит предложить решить задачу вручную, как они немедленно изобретают нечто быстрое (и это понятно: кому же захочется нагружать себя лишней работой?).

Итак, попробуйте решить сложную задачу на интуитивном уровне с реальными данными. Часто это оказывается проще сделать с большим примером.

Метод оптимизации 3: упрощение и обобщение

Метод упрощения и обобщения состоит из нескольких шагов. Сначала упрощается или изменяется некоторое ограничение (например, тип данных). После этого решается упрощенная версия задачи. Наконец, после того как алгоритм для упрощенной задачи будет построен, попытайтесь адаптировать его для более сложной задачи.

Пример: записки с требованием выкупа часто составляются из слов, вырезанных из журнала. Как определить, могла ли записка (представленная в формате строки) быть создана из заданного журнала (другая строка)?

Чтобы упростить задачу, мы слегка изменим ее: допустим, из журнала вырезаются отдельные символы, а не слова.

Чтобы решить упрощенную задачу о записке для символов, мы просто создадим массив и подсчитаем символы. Каждая позиция в массиве соответствует одной букве. Сначала мы считаем количество вхождений каждого символа в записке, а затем проверяем по журналу, присутствуют ли в нем все необходимые символы.

В обобщенной версии алгоритма делается практически то же самое, только вместо массива со счетчиками символов создается хеш-таблица, связывающая слово со счетчиком его вхождений.

Метод оптимизации 4: базовый случай и расширение

Задача сначала решается для базового случая (например, `n = 1`), после чего мы пытаемся развивать решение от исходной точки. Добравшись до более сложных/нетривиальных случаев (часто `n = 3` или `n = 4`), мы пытаемся построить их, основываясь на предыдущих результатах.

Пример: разработайте алгоритм для вывода всех возможных перестановок символов в строке (для простоты считайте, что все символы встречаются только один раз).

Рассмотрим тестовую строку `abcdefg`.

```
Случай "a" --> {"a"}
Случай "ab" --> {"ab", "ba"}
Случай "abc" --> ?
```

Вот и первый «интересный» случай. Как сгенерировать `P("abc")`, если у нас есть ответ `P("ab")`? Итак, у нас появляется дополнительная буква («c») и нам нужно вставить ее во все возможные позиции.

```
P("abc") = вставить "c" во все позиции для всех строк P("ab")
P("abc") = вставить "c" во все позиции для всех строк {"ab","ba"}
P("abc") = соединить ({"cab", "acb", "abc"}, {"cba", "bca", bac"})
P("abc") = {"cab", "acb", "abc", "cba", "bca", bac"}
```

Закономерность ясна, теперь можно разработать общий рекурсивный алгоритм. Сгенерируйте все перестановки строки $s_1 \ldots s_n$, удалив последний символ (для $s_1 \ldots s_{n-1}$). Получив список всех перестановок $s_1 \ldots s_{n-1}$, последовательно переберите все элементы, добавляя символ s_n в каждую позицию строки.

Алгоритмы, построенные таким образом, часто приводят к естественным рекурсивным решениям.

Метод оптимизации 5: мозговой штурм структур данных

Данный способ немного хаотичен, но зачастую он срабатывает. Вы просто проходите по списку структур данных и пытаетесь использовать каждую из них. Данный подход может оказаться полезным, поскольку в некоторых случаях решение задачи, что называется, «появится на поверхности», как только будет выбрана правильная структура данных (например, дерево).

Пример: была сгенерирована и сохранена в массив последовательность случайных чисел. Как найти медиану?

Пытаемся устроить мозговой штурм и подобрать адекватную структуру данных:

- Связный список? Вероятно, нет. Связные списки плохо подходят для сортировки.
- Массив? Может быть, но у нас уже есть массив. Как хранить в нем отсортированные элементы? Это довольно сложно. Отложим эту структуру данных и вернемся к ней при необходимости.
- Бинарное дерево? Вполне возможно, бинарные деревья подходят для задач сортировки. Можно попробовать усовершенствовать дерево бинарного поиска, а вершина будет медианой. Но будьте осторожны: число элементов может оказаться четным, а медиана окажется между двумя средними элементами. Два

средних элемента не могут оказаться на вершине одновременно. Возможно, этот алгоритм подойдет, мы вернемся к нему позже.

- Куча? Куча — отличный способ для сортировки и отслеживания максимальных и минимальных значений. Это интересный выбор. Если у вас будет две кучи, можно следить за «большими» и «меньшими» частями элементов. «Большая» половина находится в минимальной куче, так что самый маленький элемент оказывается в вершине, а «меньшая» половина — в максимальной куче, так что в вершине — наибольший элемент. Такие структуры данных позволяют вам найти потенциальные медианы. Если размер куч изменился, можно повторно провести балансировку, выталкивая элементы из одной кучи в другую.

Обратите внимание, что множество задач легко решаются, если правильно выбрать структуры для используемых данных. Теперь вам нужно развить интуитивное представление о том, какой подход наиболее полезен в той или иной задаче.

Неправильные ответы

Один из самых распространенных — и опасных! — слухов по поводу собеседований гласит, что кандидаты должны правильно ответить на все вопросы.

Это не совсем так.

Во-первых, ответы на вопросы собеседования не следует делить на «правильные» и «неправильные». Когда я оцениваю то, как некий кандидат показал себя на собеседовании, я никогда не думаю: «На сколько вопросов он ответил правильно?» Меня интересует то, насколько эффективным было его итоговое решение, сколько времени ему понадобилось, насколько он нуждался в помощи и насколько чистым получился его код. Словом, учитывается целый набор показателей.

Во-вторых, ваши результаты оцениваются в сравнении с другими кандидатами. Например, если вы предоставили оптимальное решение за 15 минут, а другой кандидат ответил на более простой вопрос за 5 минут, превзошел ли он вас? Возможно, но не обязательно. Если вам задают простые вопросы, резонно предположить, что вы получите оптимальные решения относительно быстро. Но если вопросы сложны, то, скорее всего, интервьюер ожидает, что ваши ответы будут не идеальными.

В-третьих, многие вопросы слишком сложны, чтобы даже сильный кандидат мог немедленно выдать оптимальный алгоритм. На решение задач, которые задаю я, сильному кандидату обычно требуется от 20 до 30 минут.

После тысячи собеседований в Google я только один раз видела кандидата с идеальным результатом. Все остальные — включая сотни тех, кто потом получил работу, — совершали ошибки.

Если вы уже знаете ответ

Если эта задача вам уже встречалась, признайтесь интервьюеру. Он задает вопросы, чтобы проверить ваши навыки решения задач. Если ответ вам уже известен, вы лишаете его такой возможности.

Кроме того, интервьюер сочтет крайне непорядочным, если вы скроете свою осведомленность. (И наоборот, признанием вы заработаете изрядное количество баллов «за честность».)

«Идеальный» язык для собеседований

Во многих ведущих компаниях интервьюеры не уделяют особого внимания языкам. Их больше интересует то, как вы решаете задачи, а не ваше знание конкретного языка.

Впрочем, другие компании — больше ориентированные на определенные языки — захотят узнать, насколько хорошо вы программируете на конкретном языке. Если вам будет предложен выбор, вероятно, стоит остановиться на том языке, с которым вы чувствуете себя увереннее всего.

С другой стороны, если вы хорошо владеете несколькими языками, стоит учесть ряд дополнительных факторов.

Распространенность

Это не обязательно, но в идеальном случае интервьюер должен знать язык, на котором вы работаете. По этой причине распространенные языки могут оказаться более уместными.

Удобочитаемость

Даже если интервьюер не знает ваш язык программирования, он должен хотя бы в общих чертах понять суть кода. Некоторые языки воспринимаются лучше других из-за их связи с другими языками.

Например, в программе на языке Java достаточно легко разберется даже тот человек, который на этом языке не работал. У большинства специалистов есть опыт работы на языках с похожим синтаксисом (таких, как C и C++). С другой стороны, синтаксис языков вроде Scala или Objective C отличается достаточно сильно.

Потенциальные проблемы

В некоторых языках программист неизбежно сталкивается с потенциальными проблемами. Например, работа на C++ означает, что помимо обычных ошибок, встречающихся в коде, могут возникнуть проблемы с управлением памятью и указателями.

Объем кода

На некоторых языках код получается более длинным, чем на других. Например, язык Java относительно «многословен» по сравнению с Python. Сравните два фрагмента:

Python:

```
dict = {"left": 1, "right": 2, "top": 3, "bottom": 4};
```

Java:

```
HashMap<String, Integer> dict = new HashMap<String, Integer>().
dict.put("left", 1);
dict.put("right", 2);
dict.put("top", 3);
dict.put("bottom", 4);
```

Впрочем, Java-код можно немного сократить. Например, кандидат на доске может написать нечто такое:

```
HM<S, I> dict = new HM<S, I>().
dict.put("left", 1);
...      "right", 2
...      "top", 3
...      "bottom", 4
```

Ему придется объяснить сокращения, но у большинства интервьюеров возражений не будет.

Простота использования

Некоторые операции в одних языках выполняются проще, чем в других. Например, в коде Python можно очень легко вернуть несколько значений из функции. В Java для этого придется создавать новый класс. Для отдельных задач это может быть удобно.

Как и в предыдущем случае, проблема может частично решаться сокращением кода или использованием несуществующих методов. Например, если один язык предоставляет функцию для транспонирования матрицы, а в другом языке такой функции нет, это не означает, что первый язык намного лучше подходит для задач, в которых такая функция нужна. Кандидат может просто считать, что в другом языке существует аналогичный метод.

Как выглядит хороший код

Вероятно, вы уже знаете, что потенциальный работодатель хочет увидеть вашу способность писать «хороший чистый» код. Но что это означает на самом деле и как проявляется на собеседованиях?

В широком смысле хороший код обладает следующими свойствами:

- **Правильность**: код должен правильно работать для всех ожидаемых и неожиданных входных данных.
- **Эффективность**: код должен обладать максимально возможной эффективностью в отношении как временной, так и пространственной сложности. Под «эффективностью» следует понимать как асимптотическую сложность («O» большое), так и практическую эффективность. Постоянный множитель обычно исключается из записи «O» большого, но в реальной жизни он может сыграть очень важную роль.

- **Простота**: если что-то можно сделать в 10 строках вместо 100, так и поступите. Скорость написания кода разработчиком тоже очень важна.
- **Удобочитаемость**: другой разработчик, прочитавший ваш код, должен понять, что и как он делает. Удобочитаемый код содержит комментарии там, где это необходимо, но его реализация понятна и проста. Это означает, что хитроумный код со сложными поразрядными сдвигами не всегда хорош.
- **Удобство сопровождения**: код должен достаточно легко адаптироваться к изменениям в жизненном цикле продукта, не создавая проблем с сопровождением (в том числе и у его автора).

Не сдавайтесь!

Вопросы на собеседованиях порой кажутся слишком сложными, но это часть того, что проверяет интервьюер. Как вы поступите — примете вызов или спасуете перед трудностями? Очень важно, чтобы вы сделали шаг вперед и бесстрашно взялись за каверзную задачу. Помните: интервью *должны* быть сложными. Так стоит ли удивляться, если вам попадется действительно сложная задача?

VIII

После собеседования

Стоило только подумать, что после собеседования можно расслабиться, как на вас обрушиваются новые переживания и волнения. Должны ли вы согласиться с предложением? Правилен ли ваш выбор? Как отказаться от предложения? Как насчет предельных сроков? Сейчас мы поговорим о некоторых тонкостях, которые позволят вам оценить и обсудить предложение.

Реакция на предложение и на отказ

Независимо от результата — соглашаетесь вы на предложение, отклоняете его или реагируете на отказ — очень важно, как вы при этом будете действовать.

Сроки принятия решения

Когда компания делает предложение, практически всегда оговаривается срок принятия решения, обычно он составляет от 1 до 4 недель. Если вы ожидаете предложений от других компаний, можно попросить немного увеличить эти временные рамки. Обычно компании идут навстречу, если это возможно.

Отклонение предложения

Даже если работа в этой компании на данный момент вам неинтересна, возможно, спустя несколько лет вам захочется поработать в ней. В ваших интересах сформулировать отказ так, чтобы остаться с компанией в хороших отношениях.

Обязательно объясните причину отказа, убедительно и вежливо. Если вы отклоняете предложение крупной компании в пользу стартапа, то объясните, что на данный момент вы чувствуете: стартап для вас более правильный выбор. Большая компания не может превратиться в стартап, поэтому ей будет трудно возразить на вашу аргументацию.

Отказ

Получить отказ неприятно, но это не означает, что вы плохой специалист. Многие хорошие специалисты плохо справляются с собеседованиями, потому что они «не находят понимания» с интервьюерами или у них «неудачный день».

К счастью, компании обычно понимают, что система интервью не идеальна и она может привести к отклонению многих хороших специалистов. По этой причине компании часто проводят повторные собеседования с ранее отклоненным претендентом.

Воспринимайте неприятный звонок как временную неудачу, а не как приговор. Поблагодарите своего собеседника за потраченное время, объясните, что вы

разочарованы, но понимаете их позицию, и поинтересуйтесь, когда можно будет пройти собеседование повторно.

Крупные технические компании обычно не предоставляют «обратную связь» по собеседованиям, но в некоторых случаях такая связь доступна. Будет полезно задать вопрос типа: «Не посоветуете ли, над чем стоит поработать в следующий раз?»

Предложение работы

Поздравляю, вы получили предложение о работе! А если повезет, то, возможно, даже несколько предложений. Задача вашего агента по подбору кадров — убедить вас принять его предложение. Как узнать, подойдет ли вам эта компания? Давайте оценим.

Финансовый пакет

Наибольшая ошибка при оценке компании, которую допускают люди, заключается в том, что они смотрят в первую очередь на зарплату. Кандидат видит эти магические цифры и принимает предложение, которое оказывается *не лучшим* в финансовом плане. Зарплата — это только верхушка айсберга вашей финансовой компенсации. Не забудьте, что еще существуют:

- *премии, оплата переезда и всевозможные льготы.* Многие компании выплачивают премии и предлагают своим сотрудникам всевозможные льготы. Когда вы оцениваете компанию, не забудьте подсчитать эти бонусы, беря в расчет как минимум три года (или срок, который вы планируете работать в этой компании);
- *стоимость проживания (местный прожиточный минимум).* Если вы получили несколько предложений из разных городов, не упускайте из виду стоимость проживания. Например, жизнь в Силиконовой долине обойдется на 30% дороже, чем в Сиэтле;
- *ежегодная премия.* Ежегодная премия в IT-компании может варьироваться от 3 до 30 %. Ваш рекрутер может сообщить вам такую информацию, но если он отказывается, поищите знакомых, работающих в этой компании;
- *фондовые опционы и гранты.* Акции являются большей частью ежегодной компенсации. Подобно льготам и премиям, расчет компенсации в акциях нужно производить на трехлетний период, а затем полученное значение добавлять к вашей зарплате.

Однако следует помнить, что также нужно учитывать перспективы карьерного роста, а это может иметь большее значение для ваших долгосрочных финансовых планов, чем зарплата. Хорошенько подумайте, насколько вам важна непосредственно озвученная заработная плата.

Карьерный рост

Каким бы заманчивым ни казалось полученное предложение, вполне возможно, что уже через несколько лет вы будете снова готовиться к собеседованию. Именно

поэтому очень важно, чтобы вы подумали уже сейчас о том, как это предложение повлияет на вашу карьеру. Задайте себе следующие вопросы:

- Насколько хорошо будет выглядеть название компании в моем резюме?
- Как я буду учиться? Чему я научусь?
- Есть ли у меня перспективы? Как развивается карьера разработчика?
- Если я собираюсь получить управленческую должность, подходит ли для этого данная компания?
- Каковы перспективы роста у этой компании (или команды)?
- Если я захочу уйти из компании, есть ли неподалеку офисы других интересных компаний или мне опять придется переезжать?

Последний пункт очень важен, но обычно его упускают из виду. Если в городе нет других IT-компаний, ваши возможности будут ограничены и вам будет труднее найти действительно хорошую вакансию.

Стабильность компании

Конечно, при равенстве других факторов стабильность желательна. Никто не хочет получить неожиданное сообщение об увольнении.

С другой стороны, другие факторы на самом деле не равны. Стабильные компании часто медленнее развиваются.

Тут все зависит от случая, но я не рекомендую зацикливаться на этом аспекте. Для некоторых кандидатов стабильность не так уж важна. Сможете ли вы достаточно быстро найти новую работу? Если сможете, возможно, стоит выбрать быстро развивающуюся компанию даже при некоторой нестабильности. С другой стороны, если вы работаете в условиях визовых ограничений или не уверены в своей способности найти новый вариант, стабильность может быть более важным фактором.

Удовольствие от работы

Наконец, вы должны понять, насколько счастливы вы будете в этой компании. На это могут повлиять следующие факторы:

- *Продукт.* Если вам нравится разрабатываемый продукт — это замечательно, но для инженера существенно важнее другой фактор — команда, в которой приходится работать.
- *Руководство и коллеги.* Когда люди говорят, что им нравится или не нравится работа, их мнение часто формируется под влиянием коллектива и руководства. Вы встречались с этими людьми? Нравится ли вам с ними общаться?
- *Корпоративная культура.* Культура компании охватывает всё — от процедуры принятия решений до психологической атмосферы и структуры компании. Поговорите об этом с будущими коллегами.
- *Рабочий график.* Ознакомьтесь с рабочим графиком и нагрузкой группы, к которой вы собираетесь примкнуть. Помните, что нагрузка в условиях приближающегося дедлайна обычно гораздо выше стандартной.

Дополнительно выясните, возможен ли переход из одной команды в другую (как, например, в Google или Facebook). Сможете ли вы найти команду, в которой вам будет комфортно?

Переговоры

В конце 2010 года я прошла специальный курс, посвященный ведению переговоров. В первый день преподаватель попросил, чтобы мы представили, что хотим купить автомобиль. Дилер А продает автомобиль за $20 000, а вот с дилером Б можно поторговаться. Какой должна быть скидка, чтобы вы обратились к дилеру Б? (Ответьте быстро, не задумываясь.)

Большинство слушателей решили, что их устроила бы скидка в $750. Другими словами, они были согласны торговаться час, чтобы сэкономить всего лишь $750. Неудивительно, что большинство студентов не могли вести переговоры с работодателем. Они соглашались на то, что предлагала компания.

Сделайте себе подарок — поторгуйтесь. Вот несколько подсказок, которые помогут вам начать переговоры.

1. *Просто начните торговаться.* Да, я знаю, что это страшно; (почти) никто не любит торговаться. Но специалисты по подбору кадров не будут отзывать предложение только потому, что вы пытаетесь добиться для себя некоторых преимуществ, поэтому вам нечего терять. Эта рекомендация особенно справедлива в том случае, если предложение поступило от крупной компании, — вам не придется торговаться со своими будущими коллегами.
2. *Всегда имейте запасной вариант.* Компании охотнее идут на уступки, если знают, что вы можете не принять предложение, а это возможно, если у вас есть выбор.
3. *Задавайте конкретные «рамки».* Вы скорее достигнете цели, если попросите увеличить зарплату на конкретные $7000, чем просто потребовав ее увеличить, — вам могут предложить $1000 и формально удовлетворить ваши требования.
4. *Завышайте требования.* При переговорах люди не всегда получают желаемое. Просите больше, чем надеетесь получить, потому что компания постарается достичь компромисса где-то посередине.
5. *Думайте не только о зарплате.* Компании часто готовы пойти на другие уступки, кроме зарплаты, поскольку если они повысят зарплату вам, то может оказаться, что ваши коллеги получают меньше. Просите увеличенную премию или какой-нибудь другой бонус. Вы, например, можете добиться выплаты пособия на переезд наличными вместо оплаты счетов. Это выгодно для студентов, у которых расходы на переезд относительно невелики.
6. *Выбирайте удобный способ ведения переговоров.* Многие советуют вести переговоры только по телефону. В какой-то степени они правы — это очень удобно. Но если вы волнуетесь, используйте электронную почту. Тот факт, что вы пытаетесь торговаться, важнее того, что вы делаете это определенным способом.

Кроме того, если вы ведете переговоры с крупной компанией, то в ней могут существовать уровни градации служащих. Всем служащим одного уровня платят одинаково. Например, в Microsoft существует такая система уровней. Вы можете торговаться в пределах зарплаты служащих вашего уровня, но для большего требуется повышение уровня. Если вы претендуете на большее, придется убедить нанимателя и будущую команду, что ваш опыт соответствует более высокому уровню, — задача трудная, но выполнимая.

На работе

Вы прошли собеседование и вздохнули с облегчением — всё позади. Наоборот, всё только начинается. Как только вы попали в компанию, пора думать о дальнейшей карьере. Куда вы уйдете и чего добьетесь в другом месте?

Создайте график своего карьерного роста

Самая обычная история — вы пришли в новую компанию и, естественно, в восторге. Все кажется таким грандиозным и перспективным, но проходит пять лет, а вы все еще на том же месте. Только тогда вы начинаете понимать, что за последние три года ничего не изменилось (ни ваши навыки, ни резюме). Так почему вы не ушли три года назад?

Когда работа доставляет удовольствие, очень легко забыться и перестать понимать, что ваша карьера забуксовала. Чтобы такого не случилось, следует представлять свою дальнейшую карьеру (хотя бы в общих чертах) прежде, чем браться за новую работу. Где вы хотите оказаться через десять лет? Что для этого нужно сделать? Кроме того, продумайте планы на следующий год и оцените прошедший: какой опыт вы получите в следующем году, как ваша карьера и ваши навыки продвинулись в прошлом году. Заранее продумывайте перспективы и регулярно проверяйте свои планы, это позволит вам избежать карьерного застоя.

Устанавливайте прочные отношения

В новой компании очень важен круг знакомств. В конце концов, личная рекомендация всегда лучше подачи заявки по Интернету.

Установите дружеские отношения со своим непосредственным руководителем и коллегами. Даже если вы уйдете из компании, поддерживайте контакты с ними. Даже дружеская записка через несколько недель после ухода поможет превратить рабочие отношения в дружеские.

Этот же подход относится и к личной жизни. Ваши друзья, друзья ваших друзей — ценные связи. Будьте готовы оказать помощь, тогда помогут и вам.

Спросите себя, что вам нужно

Некоторые руководители могут помочь вам продвинуться по карьерной лестнице, другие проявляют иной подход, основанный на невмешательстве. Только вы сами сможете решить задачи, являющиеся ключевыми для вашей карьеры.

Будьте откровенны со своим руководством. Если хотите в первую очередь заниматься техническим программированием, то скажите об этом. Если хотите стать ведущим разработчиком, обсудите с менеджером и этот аспект.

Только так вы сможете достигнуть целей согласно своему личному графику.

Проходите собеседования

Поставьте себе цель проходить собеседование хотя бы раз в год, даже если вы не ищете новую работу. Так вы сможете поддерживать свои навыки собеседований на должном уровне и будете в курсе возможных вакансий (и зарплат).

Если вы получили предложение, никто не заставляет вас принимать его. В любом случае вы сформируете связь, через которую компания, предложившая работу, сможет связаться с вами в будущем.

Часть IX

ВОПРОСЫ СОБЕСЕДОВАНИЯ

Присоединяйтесь к нам на www.CrackingTheCodingInterview.com, просматривайте код решений на других языках программирования, загружайте полные решения, обсуждайте задачи из этой книги с другими пользователями, задавайте вопросы, сообщайте об обнаруженных ошибках, просматривайте списки найденных опечаток и обращайтесь за дополнительными советами.

1

Массивы и строки

Надеюсь, что все читатели знакомы с массивами и строками, поэтому мы не будем утомлять вас подробностями. Лучше мы сосредоточимся на некоторых общих методах и тонкостях работы с этими структурами данных.

Обратите внимание, что вопросы, относящиеся к массивам и к строкам, часто эквивалентны. Таким образом, любой вопрос о массиве может быть задан применительно к строке, и наоборот.

Хеш-таблицы

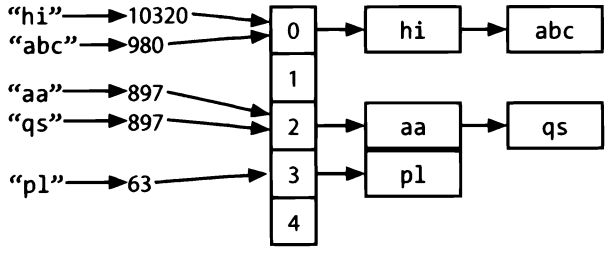
Хеш-таблица — это структура данных, связывающая ключи с ассоциированными значениями для повышения эффективности поиска. Рассмотрим простую, но весьма типичную реализацию.

В этой простой реализации используется массив связанных списков и хеш-функция. Вставка ключа (который может быть строкой или практически любым типом данных) и значения выполняется следующим образом:

1. Сначала вычисляется хеш-код ключа — обычно значение типа `int` или `long`. Учтите, что двум разным ключам могут соответствовать одинаковые хеш-коды, так как существует бесконечное количество ключей, а количество значений `int` конечно.
2. Затем хеш-коду ставится в соответствие индекс массива. Например, для этого можно воспользоваться формулой вида `hash(`*ключ*`)%`*длина_массива*. Конечно, двум разным хеш-кодам при этом может соответствовать один индекс.
3. В элементе с полученным индексом хранится связанный список ключей и значений. Ключ и значение сохраняются в элементе. Связанный список используется из-за возможных *коллизий*: двум ключам может соответствовать один хеш-код или двум разным хеш-кодам может соответствовать один индекс.

Для получения значения по заданному ключу этот процесс повторяется. Сначала по ключу вычисляется хеш-код, после чего по хеш-коду вычисляется индекс элемента. Далее в связанном списке находится значение, соответствующее ключу.

При очень высоком количестве коллизий алгоритм в худшем случае работает за время $O(N)$, где N — количество ключей. Но в общем случае предполагается хорошая реализация, сводящая количество коллизий к минимуму; тогда время поиска по ключу составляет $O(1)$.



Также хеш-таблица может быть реализована на базе сбалансированного бинарного дерева поиска. В этом случае обеспечивается время поиска $O(log\ N)$. Преимуществом такого решения является снижение затрат памяти, так как нам не придется создавать в памяти большой массив. Также появляется возможность упорядоченного перебора ключей, что иногда бывает полезно.

ArrayList и динамические массивы

В некоторых языках программирования массивы (которые в этом случае часто называются *списками*) автоматически изменяют размеры. Массив или список увеличивается при добавлении новых элементов. В других языках (например, в Java) массивы имеют фиксированную длину: их размер определяется в момент создания.

Если вам понадобится аналог массива с поддержкой динамического изменения размеров, обычно следует использовать `ArrayList`. `ArrayList` представляет собой массив, который изменяется в размерах по мере надобности и при этом обеспечивает время доступа $O(1)$. В типичной реализации при заполнении массив увеличивается вдвое. Каждое удвоение требует времени $O(n)$, но выполняется настолько редко, что амортизированное время вставки все равно составляет $O(1)$.

```
1  ArrayList<String> merge(String[] words, String[] more) {
2    ArrayList<String> sentence = new ArrayList<String>();
3    for (String w : words) sentence.add(w);
4    for (String w : more) sentence.add(w);
5    return sentence;
6  }
```

Знать эту структуру данных практически необходимо при прохождении собеседований. Обязательно освойте работу с массивами/списками с динамически изменяемыми размерами в том языке, на котором вы будете работать. Учтите, что имя структуры данных, а также «коэффициент увеличения» (равный 2 в Java) могут быть другими.

Почему амортизированное время вставки равно O(1)?

Предположим, имеется массив с размером N. Чтобы вычислить, сколько элементов должно копироваться при каждом увеличении емкости массива, проще всего пойти в обратном направлении. Заметим, что при увеличении массива до K элементов предыдущий размер массива составлял половину от нового. Следовательно, на этом шаге придется скопировать $K/2$ элементов.

```
последнее увеличение емкости: копируются n/2 элементов
предыдущее увеличение емкости: копируются n/4 элементов
предыдущее увеличение емкости: копируются n/8 элементов
...
второе увеличение емкости: копируются 2 элемента
первое увеличение емкости: копируется 1 элемент
```

Таким образом, общее количество операций копирования для вставки N элементов составляет примерно $N/2 + N/4 + N/8 + ... + 2 + 1$, что немногим менее N.

Если оценка суммы ряда не очевидна, представьте: вам нужно пройти расстояние в километр. Вы проходите 0,5 километра, потом еще 0,25 километра, потом 0,125 километра и т. д. Пройденное расстояние не превысит 1 километр (хотя и будет очень близким к этой границе).

Следовательно, общий объем работы по вставке N элементов составит $O(N)$. Каждая вставка в среднем выполняется за время $O(1)$, хотя отдельные вставки в худшем случае потребуют времени $O(N)$.

StringBuilder

Допустим, вам потребовалось объединить строки из некоторого списка. Как оценить время выполнения этого кода? Для простоты будем считать, что список состоит из n строк одинаковой длины x.

```
1  String joinWords(String[] words) {
2    String sentence = "";
3    for (String w : words) {
4      sentence = sentence + w;
5    }
6    return sentence;
7  }
```

При каждой конкатенации создается новая копия строки, куда копируются посимвольно две строки. При первой итерации будет скопировано x символов. Вторая итерация скопирует $2x$ символов, третья — $3x$. В итоге время выполнения можно оценить как $O(x + 2x + ... + nx) = O(xn^2)$.

Почему $O(xn^2)$? Потому что $1 + 2 + ... + n = n(n+1)/2$, или $O(n^2)$

Класс `StringBuilder` помогает справиться с проблемой. `StringBuilder` просто создает динамический массив всех строк и копирует их в буфер только тогда, когда это необходимо.

```
1  String joinWords(String[] words) {
2    StringBuilder sentence = new StringBuilder();
3    for (String w : words) {
4      sentence.append(w);
5    }
6    return sentence.toString();
7  }
```

Хорошее упражнение по работе со строками, массивами и структурами данных вообще — самостоятельная реализация `StringBuilder`, `HashTable` и `ArrayList`.

Вопросы собеседования

1.1. Реализуйте алгоритм, определяющий, все ли символы в строке встречаются только один раз. А если при этом запрещено использование дополнительных структур данных?

Подсказки: 44, 117, 132

1.2. Для двух строк напишите метод, определяющий, является ли одна строка перестановкой другой.

Подсказки: 1, 84, 122, 131

1.3. Напишите метод, заменяющий все пробелы в строке символами `'%20'`. Можете считать, что длина строки позволяет сохранить дополнительные символы, а фактическая длина строки известна заранее. (Примечание: при реализации метода на Java для выполнения операции «на месте» используйте символьный массив.)

Пример:

Ввод: `"Mr John Smith    ", 13`
Вывод: `"Mr%20John%20Smith"`

Подсказки: 53, 118

1.4. Напишите функцию, которая проверяет, является ли заданная строка перестановкой палиндрома. (Палиндром — слово или фраза, одинаково читающиеся в прямом и обратном направлении; перестановка — строка, содержащая те же символы в другом порядке.) Палиндром не ограничивается словами из словаря.

Пример:

Ввод: `Tact Coa`
Вывод: `True (перестановки: "taco cat", "atco cta", и т. д.)`

Подсказки: 106, 121, 134, 136

1.5. Существуют три вида модифицирующих операций со строками: вставка символа, удаление символа и замена символа. Напишите функцию, которая проверяет, находятся ли две строки на расстоянии одной модификации (или нуля модификаций).

Пример:

```
pale,  ple  -> true
pales, pale -> true
pale,  bale -> true
pale,  bake -> false
```

Подсказки: 23, 97, 130

1.6. Реализуйте метод для выполнения простейшего сжатия строк с использованием счетчика повторяющихся символов. Например, строка `aabccccaaa` превращается в `a2b1c5a3`. Если «сжатая» строка не становится короче исходной,

то метод возвращает исходную строку. Предполагается, что строка состоит только из букв верхнего и нижнего регистра (a–z).

Подсказки: 92, 110

1.7. Имеется изображение, представленное матрицей *N*×*N*; каждый пиксел представлен 4 байтами. Напишите метод для поворота изображения на 90 градусов. Удастся ли вам выполнить эту операцию «на месте»?

Подсказки: 51, 100

1.8. Напишите алгоритм, реализующий следующее условие: если элемент матрицы *M*×*N* равен 0, то весь столбец и вся строка обнуляются.

Подсказки: 17, 74, 102

1.9. Допустим, что существует метод `isSubstring`, проверяющий, является ли одно слово подстрокой другого. Для двух строк `s1` и `s2` напишите код, который проверяет, получена ли строка `s2` циклическим сдвигом `s1`, используя только один вызов метода `isSubstring` (пример: слово `waterbottle` получено циклическим сдвигом `erbottlewat`).

Подсказки: 34, 88, 104

Дополнительные вопросы: объектно-ориентированное проектирование (7.12), рекурсия (8.3), сортировка и поиск (10.9), C++ (12.11), задачи умеренной сложности (16.8, 16.17, 16.22), сложные задачи (17.4, 17.7, 17.13, 17.22, 17.26).

Подсказки начинаются на с. 690[1] (скачайте ч. XIII на сайте изд-ва «Питер»).

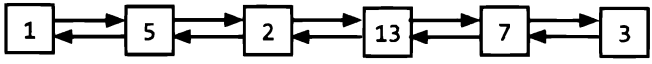
[1] Здесь и далее, чтобы получить доступ к дополнительным материалам, воспользуйтесь короткой ссылкой для скачивания http://goo.gl/ssQdRk

2

Связные списки

Связный список (linked list) – структура данных для представления совокупности *узлов*. В односвязном списке каждый узел содержит указатель на следующий узел в цепочке. В двусвязном списке в каждом узле хранятся два указателя: на следующий и предыдущий узел.

На диаграмме изображена схема двусвязного списка:



В отличие от массива, связный список не обеспечивает постоянного времени доступа к произвольному «индексу» в списке. Это означает, что если потребуется найти *K*-й элемент списка, вам придется перебрать *K* предшествующих элементов.

К преимуществам связного списка относится возможность добавления и удаления элементов в начале списка за постоянное время. В некоторых ситуациях это может быть полезно.

Создание связного списка

Ниже приведена очень простая реализация односвязного списка.

```
1  class Node {
2    Node next = null;
3    int data;
4
5    public Node(int d) {
6      data = d;
7    }
8
9    void appendToTail(int d) {
10     Node end = new Node(d);
11     Node n = this;
12     while (n.next != null) {
13       n = n.next;
14     }
15     n.next = end;
16   }
17 }
```

В этой реализации нет структуры данных `LinkedList` как таковой, а обращение к связному списку осуществляется по ссылке на начальный узел (`Node`). При такой реализации необходимо действовать осторожно: что, если ссылка на связный список будет сохранена в нескольких объектах, а потом начальный узел связного

списка изменится? Может оказаться, что в некоторых объектах хранится ссылка на старый начальный узел.

Также возможно реализовать класс `LinkedList`, инкапсулирующий класс `Node`. Фактически экземпляр этого класса будет содержать всего одну переменную: ссылку на начальный объект `Node`. Тем самым будет решена проблема, описанная выше.

Помните, что при обсуждении связных списков в ходе собеседования вы должны понимать, о каком списке идет речь: одно- или двусвязном.

Удаление узла из односвязного списка

Удаление узла из связного списка — достаточно простая задача. Для заданного узла `n` нужно найти предыдущий узел `prev` и присвоить `prev.next` значение `n.next`. Для двусвязного списка также следует обновить `n.next` и присвоить `n.next.prev` значение `n.prev`. Не забудьте: 1) выполнить проверку на `null`-указатель и 2) при необходимости обновить указатели на начало и конец списка.

Если вы пишете код на C/C++ или другом языке, в котором управлением памяти должен заниматься разработчик, проверьте, не нужно ли освободить память, которую занимал удаляемый узел.

```
1  Node deleteNode(Node head, int d) {
2    Node n = head;
3
4    if (n.data == d) {
5      return head.next; /* начальный узел изменился */
6    }
7
8    while (n.next != null) {
9      if (n.next.data == d) {
10       n.next = n.next.next;
11       return head; /* начальный узел не изменился */
12     }
13      n = n.next;
14    }
15    return head;
16  }
```

Метод бегунка

Метод бегунка (или второго указателя) используется во многих задачах по связным спискам. Связный список перебирается с использованием двух указателей одновременно, один впереди другого. «Быстрый» указатель может опережать «медленный» на фиксированное количество элементов или же может смещаться на несколько элементов для каждого узла, перебираемого «медленным» указателем.

Допустим, существует связный список $a_1->a_2->\ldots->a_n->b_1->b_2->\ldots->b_n$ и его необходимо преобразовать к виду $a_1->b_1->a_2->b_2->\ldots->a_n->b_n$. Длина связного списка неизвестна, но вы точно знаете, что она четна.

Указатель `p1` (быстрый) может смещаться на два элемента для каждого смещения `p2` на один элемент. Когда `p1` достигнет конца, `p2` будет находиться в середине

списка. Переместив `p1` обратно (в начало списка), можно провести реорганизацию. На каждой итерации `p2` выбирает элемент и вставляет его после `p1`.

Рекурсия и связные списки

Некоторые задачи о связных списках решаются с помощью рекурсии. Если у вас возникнут проблемы, подумайте, нельзя ли воспользоваться рекурсивным подходом. Сейчас мы не будем подробно говорить о рекурсии, этой теме будет посвящена отдельная глава.

Помните, что рекурсивные алгоритмы требуют затрат памяти не менее $O(n)$, где n — глубина рекурсии. Все рекурсивные алгоритмы *могут быть* заменены итерационными, однако последние могут иметь намного большую сложность.

Вопросы собеседования

2.1. Напишите код для удаления дубликатов из несортированного связного списка.

Дополнительно

Как вы будете решать задачу, если использовать временный буфер запрещено?

Подсказки: 9, 40

2.2. Реализуйте алгоритм для нахождения в односвязном списке ***k***-го элемента с конца.

Подсказки: 8, 25, 41, 67, 126

2.3. Реализуйте алгоритм, удаляющий узел из середины односвязного списка (то есть узла, не являющегося ни начальным, ни конечным — не обязательно находящегося точно в середине). Доступ предоставляется только к этому узлу.

Пример:

Ввод: узел `c` из списка `a->b->c->d->e->f`

Вывод: ничего не возвращается, но новый список имеет вид: `a->b->d->e->f`

Подсказки: 72

2.4. Напишите код для разбиения связного списка вокруг значения ***x***, так чтобы все узлы, меньшие ***x***, предшествовали узлам, большим или равным ***x***. Если ***x*** содержится в списке, то значения ***x*** должны следовать строго после элементов, меньших ***x*** (см. далее). Элемент разбивки ***x*** может находиться где угодно в «правой части»; он не обязан располагаться между левой и правой частью.

Пример:

Ввод: `3->5->8->5->10->2->1` [значение разбивки = 5]

Вывод: `3->1->2->10->5->5->8`

Подсказки: 3, 24

2.5. Два числа хранятся в виде связных списков, в которых каждый узел представляет один разряд. Все цифры хранятся в ***обратном*** порядке, при этом младший разряд (единицы) хранится в начале списка. Напишите функцию, которая суммирует два числа и возвращает результат в виде связного списка.

Пример:

Ввод: `(7->1->6) + (5->9->2)`, то есть `617 + 295`.

Вывод: `2->1->9`, то есть `912`.

Дополнительно

Решите задачу, предполагая, что цифры записаны в прямом порядке.

Ввод: `(6->1->7) + (2->9->5)`, то есть `617 + 295`.

Вывод: `9->1->2`, то есть `912`.

Подсказки: 7, 30, 71, 95, 109

2.6. Реализуйте функцию, проверяющую, является ли связный список палиндромом.

Подсказки: 5, 13, 29, 61, 101

2.7. Проверьте, пересекаются ли два заданных (одно-)связных списка. Верните узел пересечения. Учтите, что пересечение определяется ссылкой, а не значением. Иначе говоря, если *k*-й узел первого связного списка точно совпадает (по ссылке) с *j*-м узлом второго связного списка, то списки считаются пересекающимися.

Подсказки: 20, 45, 55, 65, 76, 93, 111, 120, 129

2.8. Для кольцевого связного списка реализуйте алгоритм, возвращающий начальный узел петли.

Определение

Кольцевой связный список — это связный список, в котором указатель следующего узла ссылается на более ранний узел, образуя петлю.

Пример:

Ввод: `A->B->C->D->E->C` (предыдущий узел `C`)

Вывод: `C`

Подсказки: 50, 69, 83, 90

Дополнительные вопросы: деревья и графы (4.3), объектно-ориентированное проектирование (7.12), масштабируемость и проектирование систем (9.5), задачи умеренной сложности (16.25), сложные задачи (17.12).

Подсказки начинаются на с. 690 (скачайте ч. XIII на сайте изд-ва «Питер»)

3

Стеки и очереди

На вопросы по стекам и очередям проще отвечать, когда вы хорошо разбираетесь во всех тонкостях структур данных. Впрочем, задачи порой оказываются весьма хитроумными. Хотя в некоторых задачах используется незначительная модификация исходной структуры данных, в других случаях задание может оказаться более сложным.

Реализация стека

В некоторых ситуациях стек оказывается более удобным, чем массив. Стек использует порядок LIFO («последним вошел, первым вышел»). Его можно сравнить со стопкой тарелок: последнюю добавленную в стопку тарелку возьмут первой.

Основные операции стека:

- `pop()`: извлечь элемент с вершины стека.
- `push(item)`: добавить элемент на вершину стека.
- `peek()`: вернуть элемент, находящийся на вершине стека.
- `isEmpty()`: вернуть `true` в том, и только в том случае, если стек пуст.

В отличие от массива, стек не обеспечивает постоянного времени доступа к *i*-му элементу. С другой стороны, вставка и удаление выполняются с постоянным временем, так как они не требуют перемещения элементов.

Ниже приведен простой код реализации стека. Учтите, что стек также может быть реализован на базе связного списка, если элементы добавляются и удаляются только с одного конца.

```
1  public class MyStack<T> {
2    private static class StackNode<T> {
3      private T data;
4      private StackNode<T> next;
5
6      public StackNode(T data) {
7        this.data = data;
8     }
9    }
10
11    private StackNode<T> top;
12
13    public T pop() {
14      if (top == null) throw new EmptyStackException();
15      T item = top.data;
16      top = top.next;
```

```
17      return item;
18    }
19
20    public void push(T item) {
21      StackNode<T> t = new StackNode<T>(item);
22      t.next = top;
23      top = t;
24    }
25
26    public T peek() {
27      if (top == null) throw new EmptyStackException();
28      return top.data;
29    }
30
31    public boolean isEmpty() {
32      return top == null;
33    }
34  }
```

Стек часто эффективно работает в рекурсивных алгоритмах. Иногда требуется занести временные данные в стек в процессе рекурсии, а затем удалить их при отступлении (например, из-за того, что условие рекурсии не выполняется). Стек предоставляет интуитивно понятные средства для этого.

Стек также может применяться для итеративной реализации рекурсивного алгоритма. (Кстати, хорошее упражнение — возьмите простой рекурсивный алгоритм и реализуйте его в итеративной форме!)

Реализация очереди

Очередь использует дисциплину FIFO (первым вошел, первым вышел). Она работает по тому же принципу, что и традиционная очередь в кассу: элементы удаляются из структуры данных в том же порядке, в котором были добавлены.

Очередь поддерживает следующие основные операции:

- `add(item)`: добавить элемент в начало списка.
- `remove()`: удалить первый элемент из списка.
- `peek()`: вернуть элемент в начало очереди.
- `isEmpty()`: вернуть `true` в том, и только в том случае, если очередь пуста.

Очередь также может быть реализована на базе связного списка. По сути, эти структуры данных эквивалентны, при условии, что элементы могут добавляться и удаляться с обоих концов.

```
1  public class MyQueue<T> {
2    private static class QueueNode<T> {
3      private T data;
4      private QueueNode<T> next;
5
6      public QueueNode(T data) {
7        this.data = data;
8      }
9    }
10
```

```
11    private QueueNode<T> first;
12    private QueueNode<T> last;
13
14    public void add(T item) {
15      QueueNode<T> t = new QueueNode<T>(item);
16      if (last != null) {
17        last.next = t;
18      }
19      last = t;
20      if (first == null) {
21        first = last;
22      }
23    }
24
25    public T remove() {
26      if (first == null) throw new NoSuchElementException();
27      T data = first.data;
28      first = first.next;
29      if (first == null) {
30        last = null;
31      }
32      return data;
33    }
34
35    public T peek() {
36      if (first == null) throw new EmptyStackException();
37      return first.data;
38    }
39
40    public boolean isEmpty() {
41      return first == null;
42    }
43  }
```

Особенно легко допустить ошибку при обновлении первого и последнего узла в очереди. Лишний раз проверьте это место в своем решении.

Очереди часто применяются при поиске в ширину или в реализациях кэша.

Например, при поиске в ширину мы использовали очередь для хранения списка узлов, ожидающих обработки. Каждый раз, когда мы обрабатывали узел, смежные узлы добавлялись в конец очереди. Это позволяло нам обрабатывать узлы в порядке их просмотра.

Вопросы собеседования

3.1. Опишите, как бы вы использовали один одномерный массив для реализации трех стеков.

Подсказки: 2, 12, 38, 58

3.2. Как реализовать стек, в котором кроме стандартных функций `push` и `pop` будет поддерживаться функция `min`, возвращающая минимальный элемент? Все операции — `push`, `pop` и `min` — должны выполняться за время $O(1)$.

Подсказки: 27, 59, 78

3.3. Как известно, слишком высокая стопка тарелок может развалиться. Следовательно, в реальной жизни, когда высота стопки превысила бы некоторое пороговое значение, мы начали бы складывать тарелки в новую стопку. Реализуйте структуру данных `SetOfStacks`, имитирующую реальную ситуацию. Структура `SetOfStack` должна состоять из нескольких стеков, новый стек создается, как только предыдущий достигнет порогового значения. Методы `SetOfStacks.push()` и `SetOfStacks.pop()` должны вести себя так же, как при работе с одним стеком (то есть метод `pop()` должен возвращать те же значения, которые бы он возвращал при использовании одного большого стека).

Дополнительно

Реализуйте функцию `popAt(int index)`, которая осуществляет операцию `pop` с заданным внутренним стеком.

Подсказки: 64, 81

3.4. Напишите класс `MyQueue`, который реализует очередь с использованием двух стеков.

Подсказки: 98, 114

3.5. Напишите программу сортировки стека, в результате которой наименьший элемент оказывается на вершине стека. Вы можете использовать дополнительный временный стек, но элементы не должны копироваться в другие структуры данных (например, в массив). Стек должен поддерживать следующие операции: `push`, `pop`, `peek`, `isEmpty`.

Подсказки: 15, 32, 43

3.6. В приюте для животных есть только собаки и кошки, а работа осуществляется в порядке очереди. Люди должны каждый раз забирать «самое старое» (по времени пребывания в питомнике) животное, но могут выбрать кошку или собаку (животное в любом случае будет «самым старым»). Нельзя выбрать любое понравившееся животное. Создайте структуру данных, которая обеспечивает функционирование этой системы и реализует операции `enqueue`, `dequeueAny`, `dequeueDog` и `dequeueCat`. Разрешается использование встроенной структуры данных `LinkedList`.

Подсказки: 22, 56, 63

Дополнительные вопросы: связные списки (2.6), задачи умеренной сложности (16.26), сложные задачи (17.9).

Подсказки начинаются на с. 690 (скачайте ч. XIII на сайте изд-ва «Питер»).

4

Деревья и графы

Очень часто задачи о деревьях и графах создают больше всего сложностей у кандидатов. Поиск по дереву сложнее, чем поиск в структуре данных с линейной организацией (массиве или связном списке). Кроме того, время выполнения в наихудшем случае и среднее время могут существенно различаться, а вы должны оценить оба аспекта любого алгоритма. Также играет важную роль скорость, с которой вы сможете реализовать дерево или граф с нуля.

Так как деревья обычно лучше знакомы большинству разработчиков, чем графы (и устроены немного проще), мы начнем с деревьев. Возможно, это не совсем логично, так как дерево на самом деле является разновидностью графа.

Примечание: некоторые термины, встречающиеся в этой главе, могут отличаться от терминов из других учебников и источников. Если вы привыкли к другому названию — ничего страшного, только обязательно согласуйте все расхождения в терминологии со своим интервьюером.

Разновидности деревьев

Дерево хорошо описывается в рекурсивном виде: это структура данных, состоящая из *узлов* (nodes).

- У каждого дерева существует корневой узел. (Вообще говоря, в теории графов это условие не является строго обязательным, но оно обычно справедливо для деревьев, используемых в программировании, и особенно при проведении собеседований.)
- Корневой узел имеет нуль или более дочерних узлов.
- Каждый дочерний узел имеет нуль или более дочерних узлов и т. д.

Дерево не может содержать циклов. Узлы могут следовать в определенном порядке (но это не обязательно), их значения могут относиться к произвольному типу данных, и они могут содержать ссылки на родительские узлы (но и это не обязательно).

Очень простое определение класса узла `Node` выглядит так:

```
1  class Node {
2    public String name;
3    public Node[] children;
4  }
```

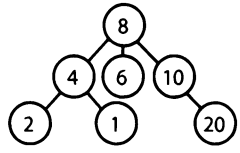
В вашем коде также может использоваться класс `Tree`, содержащий экземпляр узла. Впрочем, при проведении собеседований мы обычно обходимся без класса `Tree`. Вы можете использовать его, если вам покажется, что от этого ваш код станет проще или лучше, но такое бывает редко.

```
1  class Tree {
2    public Node root;
3  }
```

Вопросы по деревьям и графам полны неоднозначностей и неверных предположений. Обратите внимание на следующие моменты и уточните их в случае необходимости.

Деревья и бинарные деревья

Бинарным называется дерево, у которого каждый узел имеет не более двух дочерних узлов. Не все деревья являются бинарными. Например, дерево на следующей диаграмме бинарным не является (его можно было бы назвать тернарным).



Узел, не имеющий дочерних узлов, называется «листовым узлом» (или «листом»).

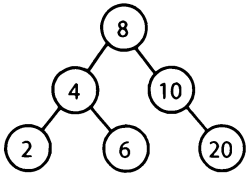
Бинарные деревья и бинарные деревья поиска

Бинарное дерево поиска представляет собой бинарное дерево, у которого для каждого узла n выполняется критерий упорядочения: все левые потомки <= n < все правые потомки.

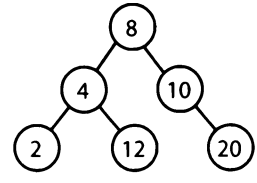
Определения бинарного дерева поиска могут слегка различаться в отношении равенства. В одних определениях дерево не может содержать дубликатов, в других дубликаты должны располагаться справа или с любой стороны. Все определения действительны, но вам стоит уточнить этот момент с интервьюером.

Следует заметить, что неравенство должно быть истинным для всех потомков узла, а не только для дочерних узлов. Скажем, левое дерево на следующей диаграмме является бинарным деревом поиска, а правое — нет, потому что 12 находится в левом поддереве 8.

Бинарное дерево поиска



Не является бинарным деревом поиска



Услышав вопрос о бинарном дереве, многие кандидаты считают, что речь идет о бинарном дереве *поиска*. Уточните, что имел в виду интервьюер. Бинарное дерево поиска предполагает, что для всех узлов левые потомки меньше или равны текущему узлу, а правые — больше текущего узла.

Сбалансированные и несбалансированные деревья

Деревья часто сбалансированы, но не всегда. Попросите интервьюера уточнить задачу. Учтите, что сбалансированность дерева не означает, что левое и правое поддеревья имеют в точности одинаковые размеры (как в случае «идеального бинарного дерева», см. ниже).

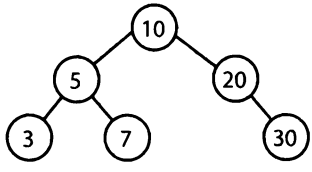
Считайте, что под «сбалансированным» деревом скорее понимается «разбалансированное в разумных пределах». Дерево сбалансировано достаточно для того, чтобы обеспечить время $O(\log n)$ для операций вставки и поиска, но не обязательно достигает максимально возможной сбалансированности.

Две распространенные разновидности сбалансированных деревьев — красно-черные деревья (с. 676) и АВЛ-деревья (с. 674) — более подробно рассматриваются в разделе «Дополнительные материалы».

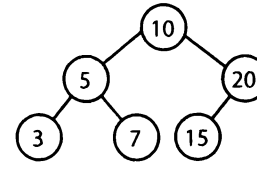
Законченные бинарные деревья

Законченным (complete) называется бинарное дерево, у которого каждый уровень (кроме последнего) содержит полный набор узлов, а последний уровень заполняется узлами слева направо.

Не является законченным бинарным деревом



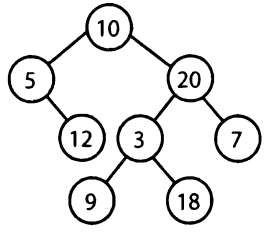
Законченное бинарное дерево



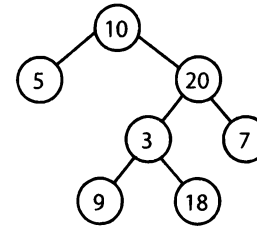
Полные бинарные деревья

У *полного* (full) бинарного дерева каждый узел имеет либо нуль, либо два дочерних узла. Иначе говоря, в полном бинарном дереве нет узлов, имеющих всего один дочерний узел.

Не является полным бинарным деревом

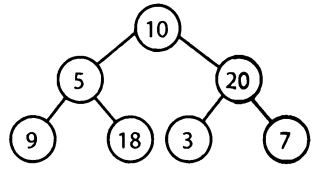


Полное бинарное дерево



Идеальные бинарные деревья

Идеальное (perfect) бинарное дерево одновременно является и полным, и законченным. Все листовые узлы находятся на одном уровне, и этот уровень содержит максимальное количество узлов.



Идеальные деревья редко встречаются как на собеседованиях, так и на практике, потому что идеальное дерево должно содержать ровно 2^k-1 узлов (где k — количество уровней). На собеседованиях не следует предполагать, что бинарное дерево является идеальным.

Обход бинарного дерева

Перед собеседованием изучите реализацию обхода дерева в префиксном, симметричном и постфиксном порядке. Наиболее распространенным является симметричный обход.

Симметричный обход

Под *симметричным* (in-order) обходом понимается обход (посещение, нередко с выводом) левого поддерева, затем текущего узла и правого поддерева.

```
1  void inOrderTraversal(TreeNode node) {
2    if (node != null) {
3      inOrderTraversal(node.left);
4      visit(node);
5      inOrderTraversal(node.right);
6    }
7  }
```

В контексте бинарного дерева поиска симметричный обход узлов происходит по возрастанию.

Префиксный обход

При *префиксном* обходе текущий узел посещается до его дочерних узлов (отсюда и название).

```
1  void preOrderTraversal(TreeNode node) {
2    if (node != null) {
3      visit(node);
4      preOrderTraversal(node.left);
5      preOrderTraversal(node.right);
6    }
7  }
```

Префиксный обход всегда начинается с корневого узла.

Постфиксный обход

При *постфиксном* обходе текущий узел посещается после его дочерних узлов (отсюда и название).

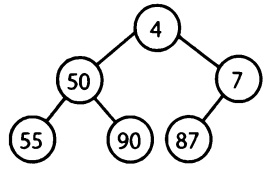
```
1  void postOrderTraversal(TreeNode node) {
2    if (node != null) {
3      postOrderTraversal(node.left);
4      postOrderTraversal(node.right);
5      visit(node);
6    }
7  }
```

При постфиксном обходе корневой узел всегда посещается в последнюю очередь.

Бинарные кучи (min-кучи и max-кучи)

Здесь мы ограничимся рассмотрением min-куч. Max-кучи фактически эквивалентны, но элементы следуют в порядке убывания, а не возрастания.

Min-куча (min-heap) представляет собой законченное бинарное дерево (то есть полностью насыщенное узлами, за исключением крайних правых узлов последнего уровня), у которого каждый узел меньше своих дочерних узлов. Таким образом, корневой узел является минимальным элементом дерева.



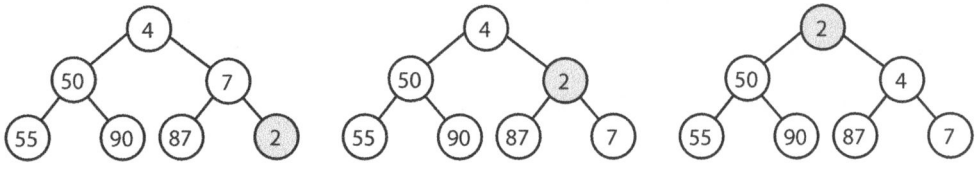
Min-куча поддерживает две важнейшие операции: вставку (`insert`) и извлечение минимума (`extract_min`).

Вставка

Вставка в min-кучу всегда начинается с нижнего уровня. Новый элемент вставляется в крайнюю правую позицию, чтобы дерево оставалось законченным.

Затем дерево «исправляется»: новый элемент меняется местами с родителем до тех пор, пока не будет найдено подходящее место для элемента. По сути, минимальный элемент «всплывает» вверх по дереву.

Шаг 1: вставить 2 Шаг 2: поменять местами 2 и 7 Шаг 3: поменять местами 2 и 4



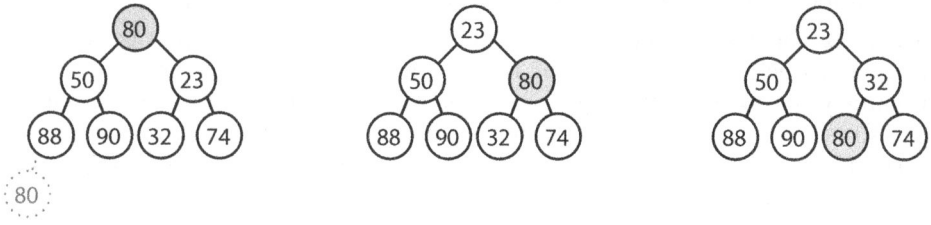
Процедура занимает время $O(\log n)$, где n — количество узлов в куче.

Извлечение минимума

Найти минимальный элемент в min-куче легко: он всегда находится на вершине. Сложнее понять, как извлечь его из дерева (хотя и это не так уж сложно).

Сначала мы удаляем минимальный элемент и меняем его местами с последним элементом в куче (крайним правым элементом на нижнем уровне). Затем элемент «опускается», меняясь местами с одним из его дочерних узлов до того момента, когда свойство min-кучи будет восстановлено.

С каким дочерним узлом он должен меняться местами — левым или правым? Это зависит от их значений. Между левым и правым элементами нет естественной упорядоченности, однако для сохранения свойства min-кучи следует выбрать меньший элемент.



Алгоритм также выполняется за время $O(\log n)$.

Нагруженные (префиксные) деревья

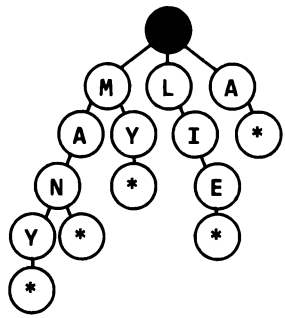
Нагруженное дерево (trie) (также иногда называемое *префиксным деревом*) — весьма интересная структура данных. Нагруженные деревья часто встречаются в вопросах на собеседованиях, но учебники по алгоритмам обычно не уделяют им особого внимания.

Нагруженное дерево представляет собой разновидность n-арного дерева, у которого в каждом узле хранятся символы. Каждый путь в дереве может представлять слово.

Узлы * (иногда называемые «нуль-узлами») часто используются для обозначения законченных слов. Например, тот факт, что за цепочкой MANY следует узел *, означает, что MANY является законченным словом. С другой стороны, существование пути MA указывает лишь на то, что существуют слова, начинающиеся с MA.

Непосредственной реализацией узлов * может быть специальная разновидность дочернего узла (например, класс `TerminatingTrieNode`, производный от `TrieNode`). Также возможно просто добавить флаг `terminates` в «родительский» узел.

Узел в нагруженном дереве может иметь от 0 до `ALPHABET_SIZE`+1 дочерних узлов.



Нагруженное дерево очень часто используется для хранения слов английского языка для ускорения поиска по префиксу. Хеш-таблица позволяет быстро проверить, является ли строка действительным словом, но она не может сообщить, является ли строка префиксом других действительных слов. Нагруженное дерево способно очень быстро выполнить такую проверку.

Насколько быстро? Нагруженное дерево проверяет строку на действительный префикс за время $O(K)$, где K – длина строки. По сути, это то же время, которое обеспечивается хеш-таблицей. Хотя о поиске по хеш-таблице часто говорят, что он выполняется за время $O(1)$, это не совсем так. Хеш-таблица должна прочитать все символы во входных данных, что потребует времени $O(K)$.

Во многих задачах со списками допустимых слов можно воспользоваться нагруженным деревом для оптимизации. В ситуациях с повторным поиском по взаимосвязанным префиксам (например, сначала M, затем MA, затем MAN и, наконец, MANY) можно передать ссылку на текущий узел дерева. Тогда вы сможете просто проверить, является ли Y дочерним узлом MAN, вместо того чтобы каждый раз начинать от корня дерева.

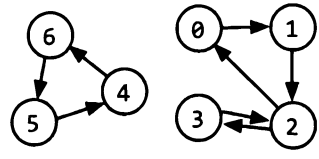
Графы

Дерево в действительности является разновидностью графа, но не каждый граф является деревом. Попросту говоря, дерево представляет собой связный граф без циклов.

Граф определяется как совокупность узлов, часть которых соединены ребрами.

- Графы делятся на направленные (как граф на схеме) и ненаправленные.
- Граф может состоять из нескольких изолированных подграфов. Если для каждой пары вершин существует путь, соединяющий эти вершины, такой граф называется *связным*.
- Граф может содержать циклы (хотя это не обязательно). *Ациклический* граф не содержит циклов.

Граф можно наглядно представить следующим образом:



В программировании существуют два основных способа представления графов.

Список смежности

Это самый распространенный способ представления графа. В каждой вершине (узле) хранится список вершин, смежных с этой вершиной. Таким образом, в ненаправленном графе ребро (a,b) будет храниться дважды: среди смежных вершин a и среди смежных вершин b.

Простое определение класса для представления узла графа почти не отличается от узла дерева.

```
1  class Graph {
2    public Node[] nodes;
3  }
4
5  class Node {
6    public String name;
7    public Node[] children;
8  }
```

Класс `Graph` необходим из-за того, что, в отличие от дерева, доступность всех узлов графа из одного узла не гарантирована.

Представление графа не обязательно требует использования дополнительных классов. Для хранения списка смежности может использоваться массив (или хеш-таблица) списков (массивов, списков массивов, связных списков и т. д.). Приведенный выше граф может быть представлен в следующем виде:

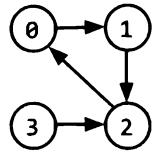
```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```

Такое решение получается более компактным, но менее наглядным. Мы предпочитаем использовать классы узлов, если только не существует веских причин для обратного.

Матрицы смежности

Матрица смежности представляет собой булеву матрицу $N \times N$ (где N — это количество узлов); истинное значение элемента `matrix[i][j]` означает, что в графе существует ребро из узла i в узел j. (Также можно использовать целочисленную матрицу с 0 и 1).

В ненаправленном графе матрица смежности симметрична, тогда как в направленном графе она (гарантированно) несимметрична.



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

С матрицами смежности могут использоваться те же алгоритмы, что и для списков смежности (поиск в ширину и т. д.), но в этом представлении они могут работать менее эффективно. Список смежности позволяет легко перебрать соседей узла, тогда как с матрицей смежности для определения соседей придется перебрать все узлы.

Поиск в графе

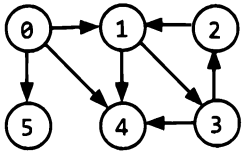
Два основных метода поиска в графе — *поиск в глубину* и *поиск в ширину*.

Поиск в глубину (DFS, Depth-First Search) начинается с корня (или любого другого произвольно выбранного узла). Алгоритм проходит до конца каждую ветвь, прежде чем переходить к следующей ветви. Таким образом, прежде чем смещаться по горизонтали, мы сначала проходим на максимальную глубину (отсюда и название).

Поиск в ширину (BFS, Breadth-First Search) также начинается с корня (или любого другого произвольно выбранного узла), но в этом случае алгоритм перебирает всех соседей, прежде чем переходить к их дочерним узлам.

Ниже приведено изображение графа и последовательности обхода в глубину и в ширину (предполагается, что соседи перебираются в числовом порядке).

Граф



Поиск в глубину

```
1  Узел 0
2   Узел 1
3    Узел 3
4     Узел 2
5     Узел 4
6  Узел 5
```

Поиск в ширину

```
1  Узел 0
2  Узел 1
3  Узел 4
4  Узел 5
5  Узел 3
6  Узел 2
```

Алгоритмы поиска в ширину и в глубину обычно используются в разных ситуациях. Поиск в глубину предпочтителен в том случае, если вы собираетесь посетить каждый узел в главе. Оба алгоритма работают, но поиск в глубину реализуется чуть проще.

Но если вы хотите найти кратчайший путь (или вообще любой путь) между двумя узлами, поиск в ширину обычно лучше. Предположим, у вас имеется граф, представляющий все дружеские отношения во всем мире, и вы пытаетесь найти путь дружеских отношений между *A* и *V*.

При поиске в глубину можно было бы пойти по пути *A->B->C->D->E->F->G-*... и уйти очень далеко, через полмира, не осознав того, что на самом деле *A* и *V* являются непосредственными друзьями. Путь все равно будет найден, но поиск займет намного больше времени. Также нет никаких гарантий, что найденный путь будет кратчайшим.

Поиск в ширину будет оставаться рядом с *A* так далеко, насколько это возможно. Алгоритм сначала переберет всех друзей *A* и перейдет к более отдаленным связям только в случае необходимости. Если *V* является другом *A* или другом его друга, такие короткие пути будут обнаружены относительно быстро.

Поиск в глубину

Алгоритм поиска в глубину сначала посещает узел *a*, а затем перебирает всех соседей *a*. При посещении узла *b*, который является соседом *a*, алгоритм сначала посещает всех соседей *b*, прежде чем переходить к другим соседям *a*. Таким образом, *a* полностью обследует ветвь *b*, прежде чем переходить к другим соседям.

Следует заметить, что префиксный обход и другие разновидности обхода дерева являются разновидностью поиска в глубину. Принципиальное различие заключается в том, что при реализации алгоритма для графа необходимо проверить, посещался ли узел ранее. Если этого не сделать, возникает опасность зацикливания.

Ниже приведена реализация поиска в глубину на псевдокоде.

```
1  void search(Node root) {
2    if (root == null) return;
3    visit(root);
4    root.visited = true;
5    for each (Node n in root.adjacent) {
6      if (n.visited == false) {
7        search(n);
8     }
9    }
10 }
```

Поиск в ширину

Поиск в ширину (BFS) менее очевиден, и у большинства кандидатов при первом знакомстве он вызывает затруднения. Главная проблема связана с предположением о том, что поиск в ширину рекурсивен. На самом деле это не так, а реализация использует очередь.

При поиске в ширину все соседи *a* проверяются до посещения любых из *их* соседей. Можно считать, что поиск ведется уровень за уровнем, начиная с *a*. Обычно в этом случае лучше всего работает итеративное решение с очередью.

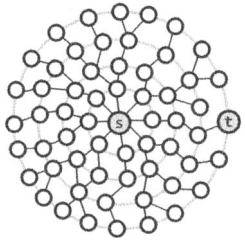
```
1  void search(Node root) {
2    Queue queue = new Queue();
3    root.marked = true;
4    queue.enqueue(root); // Добавление в конец очереди
5
6    while (!queue.isEmpty()) {
7      Node r = queue.dequeue(); // Удаление в начале очереди
8      visit(r);
9      foreach (Node n in r.adjacent) {
10       if (n.marked == false) {
11         n.marked = true;
12         queue.enqueue(n);
13     }
14    }
15   }
16 }
```

Если вам будет предложено реализовать поиск в ширину, главное — помнить об использовании очереди. Остальная часть алгоритма следует из этого факта.

Двунаправленный поиск

Двунаправленный поиск используется для поиска кратчайшего пути между начальным и конечным узлами. По сути, выполняются два одновременных поиска в ширину, по одному от каждого узла. Когда два пути поиска соприкасаются, путь найден.

Поиск в ширину
один поиск от s к t,
завершающийся через 4 уровня



Двунаправленный поиск
два поиска (один от s к t, другой от t к s),
завершающийся через 4 уровня
(по два уровня в каждом)

Чтобы понять, почему этот поиск работает быстрее, рассмотрим граф, в котором каждый узел имеет не более k смежных узлов, а кратчайший путь от узла s к узлу t имеет длину d.

- При традиционном поиске в ширину обследуются до k узлов первого «уровня» поиска. На втором уровне обследуются до k узлов для каждого из первых k узлов, так что общее количество узлов составляет k^2 (на данный момент). Это повторяется d раз, так что количество узлов составит $O(k^d)$.
- При двунаправленном поиске проводятся два поиска, которые соприкасаются после приблизительно $d/2$ уровней (середина пути). Поиск от s посещает приблизительно $k^{d/2}$ узлов, как и поиск от t. В сумме получается $2\ k^{d/2}$, или $O(k^{d/2})$.

На первый взгляд различие кажется несущественным, но это впечатление обманчиво. Различие огромно: вспомните, что $(k^{d/2})*(k^{d/2}) = k^d$. Двунаправленный поиск работает быстрее в $k^{d/2}$ раз.

Другими словами, если наша система способна поддерживать поиск путей типа «друзья друзей» при поиске в ширину, то сейчас она, скорее всего, сможет поддерживать пути «друзья друзей друзей друзей», то есть пути вдвое большей длины.

Дополнительные темы: топологическая сортировка (с. 668), алгоритм Дейкстры (с. 669), АВЛ-деревья (с. 674), красно-черные деревья (с. 676).

Вопросы интервью

4.1. Для заданного направленного графа разработайте алгоритм, проверяющий существование маршрута между двумя узлами.

Подсказки: 127

4.2. Напишите алгоритм создания бинарного дерева поиска с минимальной высотой для отсортированного (по возрастанию) массива с уникальными целочисленными элементами.

Подсказки: 19, 73, 116

4.3. Для бинарного дерева разработайте алгоритм, который создает связный список всех узлов, находящихся на каждой глубине (для дерева с глубиной D должно получиться D связных списков).

Подсказки: 107, 123, 135

4.4. Реализуйте функцию, проверяющую сбалансированность бинарного дерева. Будем считать дерево сбалансированным, если разница высот двух поддеревьев любого узла не превышает 1.

Подсказки: 21, 33, 49, 105, 124

4.5. Реализуйте функцию для проверки того, является ли бинарное дерево бинарным деревом поиска.

Подсказки: 35, 57, 86, 113, 128

4.6. Напишите алгоритм поиска «следующего» узла для заданного узла в бинарном дереве поиска. Считайте, что у каждого узла есть ссылка на его родителя.

Подсказки: 79, 91

4.7. Имеется список проектов и список зависимостей (список пар проектов, для которых первый проект зависит от второго проекта). Проект может быть построен только после построения всех его зависимостей. Найдите такой порядок построения, который позволит построить все проекты. Если действительного порядка не существует, верните признак ошибки.

Пример:

Ввод:

```
проекты: a, b, c, d, e, f
зависимости: (d, a), (b, f), (d, b), (a, f), (c, d)
```

Вывод:

```
f, e, a, b, d, c
```

Подсказки: 26, 47, 60, 85, 125, 133

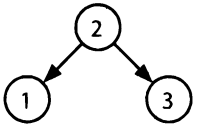
4.8. Создайте алгоритм и напишите код поиска первого общего предка двух узлов бинарного дерева. Постарайтесь избежать хранения дополнительных узлов в структуре данных. Примечание: бинарное дерево не обязательно является бинарным деревом поиска.

Подсказки: 10, 16, 28, 36, 46, 70, 80, 96

4.9. Бинарное дерево поиска было создано обходом массива слева направо и вставкой каждого элемента. Для заданного бинарного дерева поиска с разными элементами выведите все возможные массивы, которые могли привести к созданию этого дерева.

Пример:

Ввод:



Вывод: `{2, 1, 3}, {2, 3, 1}`

Подсказки: 39, 48, 66, 82

4.10. T1 и T2 — два очень больших бинарных дерева, причем T1 значительно больше T2. Создайте алгоритм, проверяющий, является ли T2 поддеревом T1.

Дерево T2 считается поддеревом T1, если существует такой узел *n* в T1, что поддерево, «растущее» из *n*, идентично дереву T2. (Иначе говоря, если вырезать дерево в узле *n*, оно будет идентично T2.)

Подсказки: 4, 11, 18, 31, 37

4.11. Вы пишете с нуля класс бинарного дерева поиска, который помимо методов вставки, поиска и удаления содержит метод getRandomNode() для получения случайного узла дерева. Вероятность выбора всех узлов должна быть одинаковой. Разработайте и реализуйте алгоритм getRandomNode; объясните, как вы реализуете остальные методы.

Подсказки: 42, 54, 62, 75, 89, 99, 112, 119

4.12. Дано бинарное дерево, в котором каждый узел содержит целое число (положительное или отрицательное). Разработайте алгоритм для подсчета всех путей, сумма значений которых соответствует заданной величине. Обратите внимание, что путь не обязан начинаться или заканчиваться в корневом или листовом узле, но он должен идти вниз (переходя только от родительских узлов к дочерним).

Подсказки: 6, 14, 52, 68, 77, 87, 94, 103, 108, 115

Дополнительные вопросы: рекурсия (8.10), масштабируемость и проектирование систем (9.2, 9.3), сортировка и поиск (10.10), сложные задачи (17.7, 17.12, 17.13, 17.14, 17.17, 17.20, 17.22, 17.25).

Подсказки начинаются на с. 690 (скачайте ч. XIII на сайте изд-ва «Питер»).

5

Операции с битами

Операции с битами применяются при решении самых разных задач. Иногда в задании явно указывается, что нужно произвести обработку на уровне отдельных битов; в других случаях это просто полезное средство оптимизации кода. Вы должны уверенно выполнять вычисления с битами как вручную, так и в коде. Будьте внимательны: в таких вычислениях очень легко совершить ошибку.

Расчеты на бумаге

Если вы подзабыли, как работает двоичная арифметика, попробуйте выполнить следующие упражнения вручную. Примеры в третьем столбце могут быть выполнены как вручную, так и с помощью «трюков», описанных ниже. Для простоты считайте, что мы работаем с четырехразрядными числами.

Если вы запутались, попробуйте взглянуть на эти операции как на операции с обычными числами в десятичной системе счисления, а затем примените этот же подход к бинарной арифметике.

`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~0 << 2)`

Решение: строка 1 (`1000`, `1111`, `1100`); строка 2 (`0101`, `1001`, `1100`); строка 3 (`0011`, `0011`, `1111`); строка 4 (`0010`, `1000`, `1000`).

Трюки для третьего столбца:

- `0110 + 0110 = 0110 * 2`, что эквивалентно смещению `0110` влево на `1`.
- Поскольку `0100 = 4`, можно умножить `0011` на `4`, что эквивалентно сдвигу влево на два бита. Мы сдвигаем `0011` влево на два бита и получаем `1100`.
- Взгляните на эту операцию с точки зрения битов. Операция `XOR` для бита и его инверсии всегда дает `1`. Поэтому `a^(~a)` всегда дает последовательность единиц.
- Значение `~0` — это последовательность единиц, так что результат `~0<<2` представляет собой последовательность единиц, за которыми следуют два нуля. Операция `AND` полученного числа с другим числом приводит к сбросу правых двух битов этого числа.

Если вы не заметили эти трюки сразу, обдумайте их на уровне логики.

Биты: трюки и факты

При решении задач на битовые операции полезно знать некоторые факты. Не пытайтесь их просто запомнить; хорошенько подумайте, почему каждое из них истинно. Записи `1s` и `0s` используются для обозначения последовательностей единиц или нулей соответственно.

```
x ^ 0s = x          x & 0s = 0          x | 0s = x
x ^ 1s = ~x         x & 1s = x          x | 1s = 1s
x ^ x = 0           x & x = x           x | x = x
```

Чтобы понять эти выражения, вспомните, что эти операции выполняются бит за битом, а то, что происходит с одним битом, не отражается на других битах. Таким образом, если одно из приведенных выше утверждений истинно для отдельного бита, оно истинно и для последовательности битов.

Поразрядное дополнение и отрицательные числа

Для представления целых чисел на компьютерах обычно используется формат дополнения до 2. Положительное число использует свое обычное представление, а отрицательное число представляется дополнением своего модуля до 2 (1 в знаковом разряде — признак отрицательного значения.) Дополнение до 2 N-разрядного числа (где N — количество разрядов в представлении числа *без учета* знакового разряда) вычисляется как дополнение числа относительно 2^N.

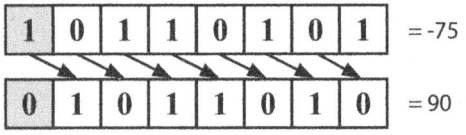
Рассмотрим представление 4-разрядного целого числа –3. В 4-разрядном числе один разряд используется для знака, а еще три — для значения. Результат должен вычисляться как дополнение исходного числа относительно 2^3, то есть 8. Дополнение 3 (модуль числа –3) относительно 8 равно 5, или `101` в двоичном представлении. Таким образом, двоичное 4-разрядное представление числа –3 равно `1101`; первый разряд в этой последовательности является знаковым.

Иначе говоря, двоичное представление отрицательного числа $-K$ в виде N-разрядного числа вычисляется по формуле `concat(1, 2^N-1-K)`.

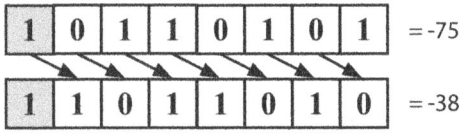
Арифметический и логический сдвиг

Существуют две разновидности операторов сдвига вправо. Арифметический сдвиг вправо фактически делит число на 2. Логический сдвиг вправо выполняет операцию, которая обычно соответствует нашим наглядным представлениям о сдвиге. Различия между ними лучше всего продемонстрировать на примере отрицательного числа.

При логическом сдвиге вправо в старший (наиболее значимый) бит помещается 0. Эта разновидность сдвига представлена оператором >>>. Для 8-разрядного целого числа (у которого знаковый бит является наиболее значимым) ситуация выглядит так, как показано на следующей диаграмме. Знаковый бит обозначен серым фоном.



При арифметическом сдвиге биты также сдвигаются вправо, но новые позиции заполняются значением из знакового бита. Общий эффект (приблизительно) эквивалентен делению на 2. Эта разновидность сдвига представлена оператором >>.



Как вы думаете, какой результат вернут следующие функции для параметров `x=-93242` и `count=40`?

```
1  int repeatedArithmeticShift(int x, int count) {
2    for (int i = 0; i < count; i++) {
3      x >>= 1; // Arithmetic shift by 1
4    }
5    return x;
6  }
7
8  int repeatedLogicalShift(int x, int count) {
9    for (int i = 0; i < count; i++) {
10     x >>>= 1; // Logical shift by 1
11   }
12   return x;
13 }
```

С логическим сдвигом результат будет равен 0, потому что в наиболее значимый бит многократно заносится 0. С арифметическим сдвигом функция вернет –1, потому что в наиболее значимый бит многократно заносится 1. Полученная последовательность, состоящая из всех 1, определяет представление целого числа –1 (со знаком).

Основные операции: получение и установка бита

Приведенные далее операции очень важны, но просто запомнить, как они работают, недостаточно. Зубрежка обязательно приведет к непоправимым ошибкам. Вместо этого разберитесь, *как* можно реализовать эти методы, тогда вы сможете успешно реализовать операции с битами — как эти, так и любые другие.

Извлечение бита

Метод `getBit` сдвигает `1` на `i` разрядов, создавая значение, например `00010000`. Операция `AND` над числом `num` сбрасывает все биты, кроме бита `i`. Затем результат сравнивается с `0`. Если новое значение отлично от `0`, значит, бит `i` равен `1`, иначе бит `i` равен `0`.

```
1  boolean getBit(int num, int i) {
2    return ((num & (1 << i)) != 0);
3  }
```

Установка бита

Метод `setBit` сдвигает `1` на `i` бит, создавая значение, например `00010000`. Операция `OR` над числом `num` изменяет только значение бита `i`. Все остальные биты остаются неизменными.

```
1  int setBit(int num, int i) {
2    return num | (1 << i);
3  }
```

Сброс бита

Этот метод может рассматриваться как противоположность методу `setBit`. Сначала создается число вида `11101111` (инверсия числа `00010000`). Затем производится операция `AND` с числом `num`. Таким образом, сбрасывается только `i`-й бит, а все остальные биты не изменяются.

```
1  int clearBit(int num, int i) {
2    int mask = ~(1 << i);
3    return num & mask;
4  }
```

Чтобы сбросить все биты от наиболее значимого до `i` (включительно), создайте маску с `1` в `i`-м бите (`1<<i`). После этого из полученной маски вычитается `1`, что дает последовательность из нулей, за которыми следуют `i` единиц. Число объединяется с маской операцией AND, и от него остаются только последние `i` бит.

```
1  int clearBitsMSBthroughI(int num, int i) {
2    int mask = (1 << i) - 1;
3    return num & mask;
4  }
```

Чтобы сбросить все биты от `i` до 0 (включительно), создайте маску из одних единиц (что равно –1) и сдвиньте ее на 31–`i` разрядов. При этом должен выполняться логический сдвиг (с продвижением знакового бита), поэтому мы используем оператор `>>>`. Так строится последовательность из единиц, за которыми следуют `i` нулей.

```
1  int clearBitsIthrough0(int num, int i) {
2    int mask = ~(-1 >>> (31 - i));
3    return num & mask;
4  }
```

Обновление бита

Чтобы присвоить `i`-му биту значение `v`, мы сначала с помощью маски (например, `11101111`) сбрасываем бит в позиции `i`. Затем нужное значение `v` сдвигается влево на `i` битов. В результате создается число, у которого `i`-й бит равен `v`, а все остальные биты нулевые. Операция `OR` для двух чисел обновляет `i`-й бит, если бит `v` равен `1`, и оставляет его нулевым в противном случае.

```
1  int updateBit(int num, int i, boolean bitIs1) {
2    int value = bitIs1 ? 1 : 0;
3    int mask = ~(1 << i);
```

```
4    return (num & mask) | (value << i);
5  }
```

Вопросы собеседования

5.1. Даны два 32-разрядных числа `N` и `M` и две позиции битов `i` и `j`. Напишите метод для вставки `M` в `N` так, чтобы число `M` занимало позицию с бита `j` по бит `i`. Предполагается, что `j` и `i` имеют такие значения, что число `M` гарантированно поместится в этот промежуток. Скажем, для `M = 10011` можно считать, что `j` и `i` разделены как минимум 5 битами. Комбинация вида `j = 3` и `i = 2` невозможна, так как число `M` не поместится между битом 3 и битом 2.

Пример:

Ввод: `N = 10000000000, M = 10011, i = 2, j = 6`
Вывод: `N = 10001001100`

Подсказки: 137, 169, 215

5.2. Дано вещественное число в интервале между 0 и 1 (например, 0,72), которое передается в формате `double`. Выведите его двоичное представление. Если для точного двоичного представления числа не хватает 32 разрядов, выведите сообщение об ошибке.

Подсказки: 143, 167, 173, 269, 297

5.3. Имеется целое число, в котором можно изменить ровно один бит из 0 в 1. Напишите код для определения длины самой длинной последовательности единиц, которая может быть при этом получена.

Пример:

Ввод: `1775  (или: 11011101111)`
Вывод: `8`

Подсказки: 159, 226, 314, 352

5.4. Для заданного положительного числа выведите ближайшие наименьшее и наибольшее числа, которые имеют такое же количество единичных битов в двоичном представлении.

Подсказки: 147, 175, 242, 312, 339, 358, 375, 390

5.5. Объясните, что делает код: `((n & (n-1)) == 0)`.

Подсказки: 151, 202, 261, 302, 346, 372, 383, 398

5.6. Напишите функцию, определяющую количество битов, которые необходимо изменить, чтобы из целого числа `A` получить целое число `B`.

Пример:

Ввод: `29 (или: 11101), 15 (или: 01111)`
Вывод: `2`

Подсказки: 336, 369

5.7. Напишите программу, меняющую местами четные и нечетные биты числа с минимальным количеством инструкций (например, меняются местами биты 0 и 1, биты 2 и 3 и т. д.).

Подсказки: 145, 248, 328, 355

5.8. Содержимое монохромного экрана хранится в одномерном массиве байтов, так что в одном байте содержится информация о восьми соседних пикселах. Ширина изображения `w` кратна 8 (то есть байты не переходят между столбцами). Высоту экрана можно рассчитать, зная длину массива и ширину экрана. Реализуйте функцию, которая рисует горизонтальную линию из точки `(x1, y)` в точку `(x2, y)`.

Сигнатура метода должна выглядеть примерно так:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

Подсказки: 366, 381, 384, 391

Дополнительные вопросы: массивы и строки (1.1, 1.4, 1.8), головоломки (6.10), рекурсия (8.4), сортировка и поиск (10.7, 10.8), C++ (12.10), задачи умеренной сложности (16.1, 16.7), сложные задачи (17.1).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

6

Головоломки

Задачи-головоломки являются очень спорной темой, многие компании запрещают использовать их на собеседованиях. Но даже если такие вопросы запрещены, это не означает, что они не могут вам достаться. Почему? Да потому, что разные фирмы по-разному понимают само понятие головоломки.

Если вам досталась головоломка, то наверняка она «честная» и ее можно решить логически. Дело вовсе не в хитроумной игре слов, решение почти всегда может быть найдено чисто логическим путем. Корни большинства головоломок лежат в математике или в информатике.

В этой главе представлены основные подходы к решению головоломок, а также та информация, которая может пригодиться при их решении.

Простые числа

Как вам, вероятно, известно, каждое положительное целое число может быть представлено в виде произведения простых чисел. Например:

$84 = 2^2*3^1*5^0*7^1*11^0*13^0*17^0*...$

Обратите внимание: показатель степени большинства простых множителей равен нулю.

Делимость

Из приведенного выше правила простых чисел следует, что для того, чтобы число `x` нацело делилось на `y` (`x\y` или `mod(y, x) = 0`), все простые числа в разложении `x` на простые множители должны присутствовать в разложении `y`. Или в более формальном виде:

Пусть `x` = $2^{J0}*3^{J1}*5^{J2}*7^{J3}*11^{J4}*...$

Пусть `y` = $2^{K0}*3^{K1}*5^{K2}*7^{K3}*11^{K4}*...$

Если `x\y`, то для всех `i` $j_i <= k_i$.

Наибольший общий делитель `x` и `y` будет равен:

`gcd(x, y)` = $2^{\min(J0, K0)} * 3^{\min(J1, K1)} * 5^{\min(J2, K2)}*...$

Наименьший общий множитель `x` и `y` будет равен:

`lcm(x, y)` = $2^{\max(J0, K0)} * 3^{\max(J1, K1)} * 5^{\max(J2, K2)} * ...$

Просто для интереса на секунду остановитесь и подумайте, что произойдет при вычислении `gcd * lcm`:

`gcd * lcm` = $2^{\min(J0, K0)} * 2^{\max(J0, K0)} * 3^{\min(J1, K1)} * 3^{\max(J1, K1)} * ...$

$= 2^{\min(J0, K0) + \max(J0, K0)} * 3^{\min(J1, K1) + \max(J1, K1)} * \ldots$

$= 2^{J0 + K0} * 3^{J1 + K1} * \ldots$

$= 2^{J0} * 2^{K0} * 3^{J1} * 3^{K1} * \ldots$

$= xy$

Проверка на простоту

Этот вопрос встречается так часто, что заслуживает особого упоминания. Наивное решение — провести перебор от 2 до n – 1 с проверкой простоты числа на каждой итерации:

```
1  boolean primeNaive(int n) {
2    if (n < 2) {
3      return false;
4    }
5    for (int i = 2; i < n; i++) {
6      if (n % i == 0) {
7        return false;
8      }
9    }
10    return true;
11  }
```

Небольшое, но важное усовершенствование — ограничение перебора до квадратного корня из n:

```
1  boolean primeSlightlyBetter(int n) {
2    if (n < 2) {
3      return false;
4    }
5    int sqrt = (int) Math.sqrt(n);
6    for (int i = 2; i <= sqrt; i++) {
7      if (n % i == 0) return false;
8    }
9    return true;
10  }
```

Порога $\sqrt{n}$ достаточно, так как для каждого числа a, на которое нацело делится n, существует дополнение b, такое что a * b = n. Если a > $\sqrt{n}$, то b < $\sqrt{n}$ (потому что $(\sqrt{n})^2 = n$). Следовательно, рассматривать a при проверке простоты n не нужно, так как это значение уже было проверено как b.

Конечно, *на самом деле* достаточно проверить, делится ли n на простое число. И в этом вам поможет решето Эратосфена.

Генерирование списка простых чисел: решето Эратосфена

Решето Эратосфена — чрезвычайно эффективный механизм генерирования списка простых чисел. Его работа основана на том факте, что каждое не простое число должно делиться на простое число.

Работа алгоритма начинается со списка всех чисел вплоть до некоторого значения max. Сначала из списка вычеркиваются все числа, делящиеся на 2. Затем алгоритм находит следующее простое число (следующее число, которое не было

вычеркнуто) и вычеркивает все числа, которые делятся на него. Последовательно вычеркивая все числа, делящиеся на 2, 3, 5, 7, 11 и т. д., мы получим список простых чисел от 2 до max.

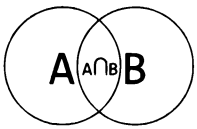
Следующий код реализует решето Эратосфена.

```
boolean[] sieveOfEratosthenes(int max) {
  boolean[] flags = new boolean[max + 1];
  int count = 0;

  init(flags); // Присвоить true всем флагам, кроме 0 и 1
  int prime = 2;

  while (prime <= Math.sqrt(max)) {
    /* Вычеркнуть оставшиеся числа, кратные prime */
    crossOff(flags, prime);

    /* Найти следующее истинное значение */
    prime = getNextPrime(flags, prime);
  }

  return flags;
}

void crossOff(boolean[] flags, int prime) {
  /* Вычеркнуть оставшиеся числа, кратные prime. Начать можно
   * с (prime*prime),  потому что значение k*prime, где k<prime,
   * уже будет вычеркнуто при предыдущей итерации. */
  for (int i = prime * prime; i < flags.length; i += prime) {
    flags[i] = false;
  }
}

int getNextPrime(boolean[] flags, int prime) {
  int next = prime + 1;
  while (next < flags.length && !flags[next]) {
    next++;
  }
  return next;
}
```

Конечно, в эту реализацию можно внести ряд оптимизаций. Простейшая оптимизация — хранение в массиве только нечетных чисел — позволит сократить затраты памяти примерно вдвое.

Вероятность

Теория вероятности может быть достаточно сложной темой, но она основана на нескольких базовых законах, которые могут быть выведены на логическом уровне.



Возьмем диаграмму Венна с представлением двух событий, A и B. Два круга представляют относительные вероятности, а область пересечения представляет событие {A и B}.

Вероятность события A и B

Представьте, что вы бросаете дротик в диаграмму Венна. Какова вероятность того, что вы попадете в область пересечения между A и B? Если вам известна вероятность попадания в A, а также процент площади A, находящейся в B (то есть вероятность нахождения в B точки, принадлежащей A), вероятность может быть выражена следующей формулой:

```
P(A и B) = P(B при условии принадлежности A) P(A)
```

Например, представьте ситуацию со случайным выбором числа от 1 до 10 (включительно). Какова вероятность того, что будет выбрано четное число в диапазоне от 1 до 5? Вероятность выбора числа от 1 до 5 равна 50 %, а вероятность четности числа в диапазоне от 1 до 5 равна 40 %. Таким образом, вероятность выполнения обоих условий равна:

```
P(x четно и x <= 5)
  = P(x четно при условии x <= 5) P(x <= 5)
  = (2/5) * (1/2)
  = 1/5
```

Заметим, что поскольку P(A и B) = P(B при условии A) P(A) = P(A при условии B) P(B), вероятность A при условии B может быть выражена через обратную вероятность:

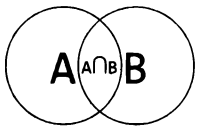
```
P(A при условии B) = P(B при условии A) P(A) / P(B)
```

Эта формула называется *теоремой Байеса*.

Вероятность A или B

Теперь представьте, что вы хотите определить вероятность попадания в A или B. Если известна вероятность попадания в каждую область по отдельности, а также известна вероятность попадания в область пересечения, искомая вероятность может быть выражена в следующем виде:

```
P(A или B) = P(A) + P(B) - P(A и B)
```



С точки зрения логики это понятно. Если просто просуммировать размеры, область пересечения будет учтена в сумме дважды. Лишнее вхождение необходимо вычесть. Это тоже можно выразить на диаграмме Венна.

Например, представьте ситуацию со случайным выбором числа от 1 до 10 (включительно). Какова вероятность того, что будет выбрано четное число *или* число в диапазоне от 1 до 5? Вероятность выбора четного числа равна 50%, а вероятность выбора числа от 1 до 5 равна 50%. Как мы уже знаем, вероятность выполнения обоих условий равна 20%. Итак, искомая вероятность равна:

```
P(x четно или x <=5)
  = P(x четно) + P(x <= 5) - P(x четно и x <= 5)
  =  1/2 + 1/2 - 1/5
  =  4/5
```

Вероятности для других особых случаев — независимых событий и взаимоисключающих событий — вычисляются просто.

Независимость

Если события A и B независимы (то есть факт возникновения одного события никак не влияет на вероятность возникновения другого), то P(A и B) = P(A) P(B). Это правило следует из того, что P(B при условии A) = P(B), так как A не влияет на B.

Взаимоисключающие события

Если события A и B являются взаимоисключающими (то есть при возникновении одного из них другое невозможно), то P(A or B) = P(A) + P(B). Это объясняется тем, что P(A and B) = 0, поэтому эта составляющая исключается из предшествующей формулы P(A или B).

Как ни странно, многие разработчики путают концепции независимости и взаимоисключаемости. Это совершенно разные понятия. Более того, два события не могут быть одновременно независимыми и взаимоисключающими (при условии, что вероятность каждого из них отлична от 0). Почему? Потому что взаимоисключаемость означает, что если одно событие произошло, то другое произойти не может. Напротив, независимость означает, что возникновение одного события никак не влияет на возникновение другого. Таким образом, если два события имеют ненулевые вероятности, они никогда не могут быть взаимоисключающими и независимыми одновременно.

Если вероятность одного или обоих событий равна нулю (то есть событие невозможно), то события одновременно являются независимыми и взаимоисключающими. Этот факт доказывается простым применением определений (то есть формул) независимости и взаимоисключаемости.

Начинайте говорить

Если вам досталась головоломка, не паникуйте. Как и в случае с задачами на алгоритмы, интервьюеры хотят видеть, как вы пытаетесь решить задачу. Они не ожидают, что вы сразу дадите ответ. Начните рассуждать вслух, покажите интервьюеру, как вы решаете задачу.

Правила и шаблоны

В большинстве случаев попробуйте найти и записать правила или шаблоны, которые помогут вам решить задачу. Да-да, именно записать — это поможет запомнить их и использовать при решении задачи. Давайте рассмотрим простой пример.

У вас две веревки и каждая из них горит ровно один час. Как их можно использовать, чтобы определить, что прошло 15 минут? Более того, плотность веревок не является константой, поэтому необязательно, что половина веревки будет гореть ровно полчаса.

Совет: остановитесь здесь и попытайтесь решить задачу самостоятельно. В крайнем случае, прочитайте этот раздел медленно и вдумчиво. Каждый абзац будет приближать вас к решению.

Из постановки задачи ясно, что у нас есть один час. Можно получить двухчасовой интервал, если поджечь вторую веревку после того, как догорит первая. Таким образом, мы получаем правило.

Правило 1: если одна веревка горит `x` минут, а другая горит `y` минут, то общее время горения составит `x+y`.

Что еще мы можем сделать с веревкой? Попытка поджечь веревку в любом другом месте, кроме как с концов, не даст нам дополнительной информации. Огонь будет расходиться в обе стороны, и мы понятия не имеем, сколько времени она будет гореть.

Но мы можем поджечь веревку с двух концов одновременно. Огонь должен будет встретиться через 30 минут.

Правило 2: если веревка горит `x` минут, мы можем отсчитать интервал времени, равный `x/2` минут.

Итак, теперь мы знаем, что можем отсчитать 30 минут, используя одну веревку. Это также означает, что мы можем вычесть 30 минут из времени горения второй веревки, если одновременно подожжем первую веревку с двух концов, а вторую только с одного.

Правило 3: если первая веревка горит `x` минут, а вторая веревка горит `y` минут, мы можем заставить вторую веревку гореть (`y-x`) минут или (`y-x/2`) минут.

Теперь давайте соединим все части воедино. Мы можем превратить вторую веревку в веревку, которая горит 30 минут. Если мы теперь подожжем вторую веревку с другого конца (см. правило 2), то она будет гореть 15 минут.

Запишем получившийся алгоритм:

- Поджигаем веревку 1 с двух сторон, веревку 2 — только с одного.
- Веревка 1 сгорела, значит, прошло 30 минут, веревке 2 осталось гореть 30 минут.
- В этот момент времени поджигаем веревку 2 с другого конца.
- Чтобы веревка 2 сгорела полностью, понадобится 15 минут.

Обратите внимание, как записанные правила облегчили решение этой задачи.

Балансировка худшего случая

Многие головоломки относятся к задачам минимизации худшего случая; в формулировке таких задач требуется *свести к минимуму* количество действий или выполнить что-либо не более заданного количества раз. Можно попробовать «сбалансировать» худший случай. Таким образом, если принимаемое на ранней стадии решение приведет к искажению результата в сторону худшего случая, можно выполнить балансировку. Рассмотрим конкретный пример.

Задача о «девяти шарах» — классика собеседования. У вас есть девять шаров — восемь имеют одинаковый вес, а один более тяжелый. Вы можете воспользоваться весами, позволяющими узнать, какой шар тяжелее. Требуется найти тяжелый шар за два взвешивания.

Разделите шары на две группы (по четыре шара), оставшийся (девятый) шар можно пока отложить в сторону. Если одна из групп тяжелее, значит, шар находится в ней. Если обе группы имеют одинаковый вес, тогда девятый шар — самый тяжелый. Если воспользоваться этим же методом еще раз, то в худшем случае вы получите результат за три взвешивания — одно лишнее!

Перед нами типичный пример несбалансированного худшего случая: если девятый шар является самым тяжелым, потребуется одно взвешивание, а во всех остальных случаях — три. Если мы «оштрафуем» девятый шар, выделив для него дополнительные шары, это позволит снизить нагрузку на другие: классический пример балансировки худшего случая.

Если мы разделим шары на группы по три шара в каждой, то достаточно одного взвешивания, чтобы понять, в какой группе находится тяжелый шар. Мы можем даже сформулировать *правило*: для N шаров, где N кратно трем, за одно взвешивание можно найти группу шаров $N/3$, в которой находится самый тяжелый шар.

Для оставшейся группы из трех шаров нужно повторить ту же операцию: отложить один шар, а остальные взвесить и выбрать более тяжелый. Если шары весят одинаково, выбирается отложенный шар.

Алгоритмический подход

Если вы не можете сразу найти решение, попробуйте использовать один из методов поиска ответа на алгоритмические вопросы (с. 61). Головоломки часто представляют собой не что иное, как те же задачи алгоритмизации, из которых убраны технические нюансы. Особенно полезны методы «базовый случай» и «сделай сам».

Дополнительная информация: Полезные формулы (с. 666).

Вопросы собеседования

6.1. Есть 20 баночек с таблетками. В 19 баночках лежат таблетки весом 1 г, а в одной — весом 1,1 г. Даны весы, показывающие точный вес. Как за одно взвешивание найти банку с тяжелыми таблетками?

Подсказки: 186, 252, 319, 387

6.2. Имеется баскетбольное кольцо, и вам предлагается сыграть в одну из двух игр:

Игра 1: У вас есть один бросок в кольцо.

Игра 2: У вас есть три попытки, но для победы потребуется не менее двух попаданий.

Если p — вероятность попадания в кольцо при броске, при каких значениях p следует выбирать ту или иную игру?

Подсказки: 181, 239, 284, 323

6.3. Имеется шахматная доска размером 8×8, из которой были вырезаны два противоположных по диагонали угла, и 31 кость домино; каждая кость домино может закрыть два квадратика на поле. Можно ли вымостить костями всю доску? Обоснуйте ответ (приведите пример или докажите, что это невозможно).

Подсказки: 367, 397

6.4. На каждой из трех вершин треугольника сидит муравей. Какова вероятность столкновения (на любой из сторон), если муравьи начнут ползти по сторонам треугольника? Предполагается, что каждый муравей выбирает направление случайным образом, вероятность выбора направлений одинакова, и все муравьи ползут с одинаковой скоростью.

Также определите вероятность столкновения для n муравьев на многоугольнике с n вершинами.

Подсказки: 157, 195, 296

6.5. У вас есть пятилитровый и трехлитровый кувшины и неограниченное количество воды. Как отмерить ровно 4 литра воды? Кувшины имеют неправильную форму, поэтому точно отмерить «половину» кувшина невозможно.

Подсказки: 149, 379, 400

6.6. На остров приезжает гонец со странным приказом: все голубоглазые люди должны как можно скорее покинуть остров. Самолет улетает каждый вечер в 20:00. Каждый человек может видеть цвет глаз других, но не знает цвет собственных (и никто не имеет права сказать человеку, какой у него цвет глаз). Жители острова не знают, сколько на нем живет голубоглазых; известно лишь то, что есть минимум один. Сколько дней потребуется, чтобы все голубоглазые уехали?

Подсказки: 218, 282, 341, 370

6.7. Королева нового постапокалиптического мира обеспокоена проблемой рождаемости. Она издает закон, по которому в каждой семье должна родиться хотя бы одна девочка. Если все семьи повинуются закону, то есть заводят детей, пока не родится девочка, после чего немедленно прекращают, — каким будет соотношение полов в новом поколении? (Предполагается, что вероятности рождения мальчика или девочки равны.) Сначала решите задачу на логическом уровне, а затем напишите компьютерную модель.

Подсказки: 154, 160, 171, 188, 201

6.8. Имеется 100-этажное здание. Если яйцо сбросить с высоты N-го этажа (или с большей высоты), оно разобьется. Если его бросить с меньшего этажа, оно не разобьется. У вас есть два яйца; найдите N за минимальное количество бросков.

Подсказки: 156, 233, 294, 333, 357, 374, 395

6.9. В длинном коридоре расположены 100 закрытых замков. Человек сначала открывает все сто. Затем он закрывает каждый второй замок. Затем он делает еще один проход — «переключает» каждый третий замок (если замок был открыт, то он его закрывает, и наоборот). Процесс продолжается 100 раз, на i-м проходе изменяется состояние каждого i-го замка. Сколько замков останутся открытыми после 100-го прохода, когда «переключается» только замок № 100?

Подсказки: 139, 172, 264, 306

6.10. Имеется 1000 бутылок лимонада, ровно одна из которых отравлена. Также у вас есть 10 тестовых полосок для обнаружения яда. Даже одна капля яда

окрашивает полоску и делает ее непригодной для дальнейшего использования. На тестовую полоску можно одновременно нанести любое количество капель, и одна полоска может использоваться сколько угодно раз (при условии, что все пробы были отрицательными). Однако вы можете проводить испытания не чаще одного раза в день, а для получения результата с момента проведения проходит семь дней. Как найти отравленную бутылку за минимальное количество дней?

Дополнительно

Напишите программную модель вашего решения.

Подсказки: 146, 163, 183, 191, 205, 221, 230, 241, 249

Дополнительные вопросы: задачи умеренной сложности (16.5), сложные задачи (17.19).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

7

Объектно-ориентированное проектирование

Задания по объектно-ориентированному проектированию (ООП) предполагают, что кандидат может спланировать набор классов и методов, позволяющих реализовать техническую задачу или смоделировать реальный объект. Эти задачи дают (или по крайней мере должны давать) интервьюеру некоторое представление о стиле программирования кандидата.

На таких задачах вы демонстрируете не столько свое знание паттернов проектирования, сколько понимание того, как написать элегантный объектно-ориентированный код, не создающий проблем с сопровождением. Провал при выполнении этих заданий может поставить под угрозу результат всего собеседования.

Как подходить к решению заданий

Независимо от того, о чем идет речь — о материальном объекте или технической задаче, — решение ООП-задания может идти по стандартному пути. Приведенный далее подход пригодится для решения многих задач.

Шаг 1. Избавьтесь от неоднозначностей

Вопросы по ООП иногда содержат специально добавленные неопределенности. Это делается для того, чтобы проверить, как вы поступите: сделаете предположения или зададите уточняющие вопросы. В конце концов, если разработчик при написании кода толком не понимает, чего от него ждут, он лишь попусту тратит время и деньги компании и может создать куда более серьезные проблемы.

Получив задание по ООП, вы должны выяснить, *кто* и *как* будет использовать ваш программный продукт. Возможно, в зависимости от вопроса в этом диалоге вам даже придется пройти через процедуру «шести W» (who, what, where, when, how, why) — кто, что, где, когда, как, почему.

Например, предположим, что вас попросили написать ООП-модель кофеварки. Задание выглядит однозначным? Увы, это не так.

Кофеварка может стоять в большом ресторане, обслуживающем сотни клиентов в час, и варить десятки разновидностей кофе. Или это может быть очень простое устройство на кухне пожилой семейной пары, способное без проблем сварить только чашечку черного кофе. Как видите, проект очень сильно зависит от конкретного сценария использования.

Шаг 2. Определите основные объекты

Разобравшись с тем, что именно будет проектироваться, необходимо выделить основные объекты системы. Если мы будем разрабатывать кофеварку для ресторана, основными объектами будут `Table`, `Guest`, `Party`, `Order`, `Meal`, `Employee`, `Server` и `Host`.

Шаг 3. Проанализируйте связи

Основные объекты выделены, теперь нужно проанализировать связи между ними. Какие члены должны присутствовать в наших объектах? Есть ли объекты, которые наследуют от каких-либо других объектов? Какие связи используются — «многие-ко-многим» или «один-ко-многим»?

В случае с ресторанной кофеваркой проект будет иметь вид:

- в `Party` присутствует массив `Guests`;
- `Server` и `Hosts` наследуют от `Employee`;
- у каждого `Table` есть один `Party`, но у каждого `Party` может быть несколько `Table`;
- для каждого `Restaurant` есть один `Host`.

Будьте очень осторожны: вы можете сделать неправильные предположения. Например, стол (`Table`) может быть общим для нескольких `Party`, — так называемый общий стол в некоторых модных тусовочных местах. Вы должны обсудить со своим интервьюером цель проекта.

Шаг 4. Исследуйте действия

К этому моменту у вас должна быть готова общая схема объектно-ориентированного проекта. Остается рассмотреть основные действия, которые могут быть выполнены объектами, и их взаимосвязи. При этом может выясниться, что вы забыли какие-либо объекты, и в проект придется вносить изменения.

Например, компания (`Party`) идет в ресторан (`Restaurant`), и гости (`Guests`) интересуются свободным столиком (`Table`) у официанта (`Host`). Официант просматривает список резервирования (`Reservation`) и, если есть свободные места, выделяет компании (`Party`) столик (`Table`). Если свободных мест нет, компания становится в очередь (в конец списка). Когда компания уходит, стол освобождается и его занимает следующая по порядку в очереди компания.

Паттерны проектирования

Поскольку интервьюеры пытаются проверить ваши возможности, а не ваши знания, паттерны проектирования чаще всего остаются за рамками собеседования. Однако паттерны «Одиночка» (Singleton) и «Фабричный метод» (Factory Method) часто встречаются в собеседованиях, поэтому мы остановимся на них поподробнее.

Существует множество паттернов, и в этой книге мы не можем рассмотреть их все. Так что если вы захотите улучшить свою квалификацию в области проектирования, найдите специализированную книгу по этой теме.

Одиночка

Паттерн «Одиночка» гарантирует, что класс существует только в одном экземпляре и обеспечивает доступ к экземпляру в коде приложения. Он может пригодиться в том случае, когда у вас есть «глобальный» объект, который должен существовать не более чем в одном экземпляре. Например, мы можем реализовать класс `Restaurant` так, чтобы существовал единственный экземпляр `Restaurant`.

```
1  public class Restaurant {
2    private static Restaurant _instance = null;
3    protected Restaurant() { ... }
4    public static Restaurant getInstance() {
5      if (_instance == null) {
6        _instance = new Restaurant();
7     }
8      return _instance;
9    }
10 }
```

Стоит отметить, что многие разработчики не любят паттерн «Одиночка» и даже относят его к «антипаттернам». Одна из причин такого отношения заключается в том, что этот паттерн может усложнить модульное тестирование.

Фабричный метод

Паттерн «Фабричный метод» предоставляет интерфейс для создания экземпляров класса; при этом субклассы выбирают класс, экземпляр которого должен быть создан. Например, класс-создатель может быть абстрактным и не предоставлять реализацию фабричного метода. Также возможно создать конкретный класс `Creator`, предоставляющий реализацию фабричного метода. В таком случае фабричный метод будет получать параметр, определяющий, экземпляр какого класса должен быть создан.

```
1  public class CardGame {
2    public static CardGame createCardGame(GameType type) {
3      if (type == GameType.Poker) {
4        return new PokerGame();
5      } else if (type == GameType.BlackJack) {
6        return new BlackJackGame();
7     }
8      return null;
9    }
10 }
```

Вопросы собеседования

7.1. Разработайте структуры данных для универсальной колоды карт. Объясните, как разделить структуры данных на субклассы, чтобы реализовать игру в блэкджек.

Подсказки: 153, 275

7.2. Имеется центр обработки звонков с тремя уровнями сотрудников: оператор, менеджер и директор. Входящий телефонный звонок адресуется свободному

оператору. Если оператор не может обработать звонок, он автоматически перенаправляется менеджеру. Если менеджер занят, то звонок перенаправляется директору. Разработайте классы и структуры данных для этой задачи. Реализуйте метод `dispatchCall()`, который перенаправляет звонок первому свободному сотруднику.

Подсказки: 363

7.3. Разработайте модель музыкального автомата, используя принципы ООП.

Подсказки: 198

7.4. Разработайте модель автостоянки, используя принципы ООП.

Подсказки: 258

7.5. Разработайте структуры данных для онлайн-библиотеки.

Подсказки: 344

7.6. Запрограммируйте модель сборной головоломки («пазл») из $N \times N$ фрагментов. Разработайте структуры данных и объясните алгоритм, позволяющий решить задачу. Предполагается, что существует метод `fitsWith`, возвращающий значение `true` в том случае, если два переданных фрагмента головоломки должны располагаться рядом.

Подсказки: 192, 238, 283

7.7. Как бы вы подошли к проектированию чат-сервера? Предоставьте информацию о компонентах внутренней подсистемы (backend), классах и методах. Перечислите самые трудные задачи, которые необходимо решить.

Подсказки: 213, 245, 271

7.8. В «реверси» играют по следующим правилам: каждая фишка в игре с одной стороны белая, а с другой — черная. Когда ряд фишек оказывается ограничен фишками противника (слева и справа или сверху и снизу), цвет фишек в этом ряду меняется на противоположный. При своем ходе игрок обязан захватить по крайней мере одну из фишек противника, если это возможно. Игра заканчивается, когда у обоих игроков не остается допустимых ходов. Побеждает тот, у кого больше фишек на поле. Реализуйте ООП-модель для этой игры.

Подсказки: 179, 228

7.9. Реализуйте класс `CircularArray` для представления структуры данных — аналога массива с эффективной реализацией циклического сдвига. Если возможно, класс должен использовать обобщенный (generic) тип и поддерживать перебор в стандартном синтаксисе (`Obj o : circularArray`).

Подсказки: 389

7.10. Спроектируйте и реализуйте текстовый вариант игры «Сапер». В этой классической компьютерной игре для одного игрока на поле $N \times N$ тайно расставляются B мин. Остальные ячейки либо пусты, либо в них выводится цифра — количество мин в окружающих восьми ячейках. Игрок открывает ячейку. Если в ней окажется мина, то игрок проигрывает. Если ячейка содержит число, то игрок видит это число. Если ячейка пуста, то открывается эта ячейка со всеми

прилегающими пустыми ячейками (до окружающих ячеек с цифрами). Игрок побеждает, если ему удастся открыть все ячейки, не содержащие мин. Также игрок может помечать некоторые ячейки как потенциальные мины. На ход игры такая пометка не влияет, она лишь предотвращает случайные щелчки на ячейках, в которых, по мнению игрока, находится мина. (Подсказка: если эта игра вам незнакома, сыграйте несколько партий в Интернете.)

Игровое поле с тремя минами. Изначально игрок не видит его.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

В начале игры выводится полностью закрытое поле.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Щелчок на ячейке (строка=1, столбец=0) открывает часть игрового поля:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

Пользователь выигрывает, если открыты все ячейки, не содержащие мин.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

Подсказки: 351, 361, 377, 386, 399

7.11. Объясните, какие структуры данных и алгоритмы вы бы использовали для разработки файловой системы, хранящейся в оперативной памяти. Напишите программный код, иллюстрирующий использование этих алгоритмов.

Подсказки: 141, 216

7.12. Спроектируйте и реализуйте хеш-таблицу, использующую связные списки для обработки коллизий.

Подсказки: 287, 307

Дополнительные вопросы: потоки и блокировки (16.3).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

8

Рекурсия и динамическое программирование

Существует огромное множество разнообразных рекурсивных задач, но большинство из них строится по похожим схемам. Подсказка: если задача рекурсивна, то она может быть разбита на подзадачи.

Совет: мой опыт обучения показывает, что интуиция «Да это похоже на рекурсивную задачу» срабатывает примерно с 50%-ной точностью. Используйте этот инстинкт, это очень полезные 50%. Но не бойтесь взглянуть на задачу под другим углом, даже если на первых порах она показалась вам рекурсивной. Существует 50%-ная вероятность того, что первое впечатление было ошибочным.

Когда вы получаете задание, начинающееся со слов «Разработайте алгоритм для вычисления *N*-го...», или «Напишите код для вывода первых *n*...», или «Реализуйте метод для вычисления всех...» — скорее всего, речь пойдет о рекурсии.

С чего начать

Рекурсивные решения по определению основываются на решении подзадач. Очень часто вам приходится вычислять `f(n)`, добавляя, вычитая или еще как-либо изменяя решение для `f(n-1)`. В других случаях задача может решаться для первой половины набора данных, а затем для второй половины с последующим слиянием результатов.

Существует много способов деления задачи на подзадачи. Три самых распространенных метода разработки алгоритма — восходящий, нисходящий и половинчатый.

Восходящая рекурсия

Восходящая рекурсия обычно более понятна на интуитивном уровне. Вы знаете, как решить задачу для самого простого случая — например, для списка всего с одним элементом. Затем задача решается для двух элементов, затем для трех и т. д. Подумайте о том, как построить решение для конкретного случая, основываясь на решении для предыдущего случая (или нескольких предыдущих случаев).

Нисходящая рекурсия

Нисходящая рекурсия выглядит более сложной, так как она менее конкретна. Тем не менее в отдельных случаях такой подход к решению задачи оказывается оптимальным.

В этом случае необходимо решить, как разделить задачу для случая N на подзадачи. Будьте осторожны с перекрывающимися случаями.

Половинчатая рекурсия

Помимо восходящей и нисходящей рекурсии также часто эффективно работает метод разбиения набора данных на две половины.

Например, алгоритм бинарной сортировки основан на «половинчатом» методе. При поиске элемента в отсортированном массиве вы сначала определяете, какая половина массива содержит искомое значение. Затем происходит рекурсивный переход, и поиск продолжается в выбранной половине.

Алгоритм сортировки слиянием также относится к «половинчатым» методам. Каждая половина массива сортируется по отдельности, после чего отсортированные половины объединяются.

Решения рекурсивные и итеративные

Рекурсивные алгоритмы могут быть весьма неэффективны по затратам памяти. Каждый рекурсивный вызов добавляет новый уровень в стек; это означает, что если алгоритм проводит рекурсию до уровня n, он использует как минимум $O(n)$ памяти.

По этой причине часто бывает лучше реализовать рекурсивный алгоритм в итеративном виде. Все рекурсивные алгоритмы могут быть реализованы в итеративном виде, хотя иногда это приводит к существенному усложнению кода. Прежде чем браться за рекурсивный код, спросите себя, насколько сложно будет реализовать его в итеративном виде, и обсудите достоинства и недостатки каждого варианта с интервьюером.

Динамическое программирование и мемоизация

Среди разработчиков распространено мнение о сложности задач динамического программирования, однако бояться их не стоит. Собственно, когда вы усвоите суть метода, такие задачи становятся очень простыми.

Методология динамического программирования в основном сводится к определению рекурсивного алгоритма и нахождению перекрывающихся подзадач. Полученные результаты кэшируются для будущих рекурсивных вызовов.

При другом подходе следует проанализировать закономерность рекурсивных вызовов и использовать итеративную реализацию. При этом по-прежнему возможно «кэширование» предшествующей работы.

Замечание по поводу терминологии: некоторые специалисты называют нисходящее динамическое программирование «мемоизацией» (memoization), а термин «динамическое программирование» используют только для восходящих методов. Мы не будем различать эти случаи, и используем термин «динамическое программирование».

Один из простейших примеров динамического программирования — вычисление n-го числа Фибоначчи. Хороший подход к задачам такого рода часто заключается в том, чтобы реализовать задачу в обычном рекурсивном виде, а затем добавить в решение кэширование.

Числа Фибоначчи

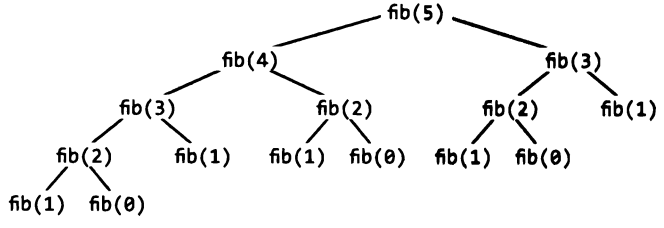
Рекурсия

Начнем с рекурсивной реализации. Просто, не так ли?

```
1  int fibonacci(int i) {
2    if (i == 0) return 0;
3    if (i == 1) return 1;
4    return fibonacci(i - 1) + fibonacci(i - 2);
5  }
```

Каково время выполнения этой функции? Немного подумайте, прежде чем отвечать.

Если вы ответили $O(n)$ или $O(n^2)$ (как отвечает большинство людей), подумайте снова. Проанализируйте последовательность выполнения кода. Попробуйте нарисовать ветви выполнения в виде дерева (то есть нарисовать дерево рекурсии) — это полезно в этой и многих других рекурсивных задачах.



Заметьте, что на всех листьях дерева находятся вызовы `fib(1)` и `fib(0)`. Они могут рассматриваться как базовые случаи.

Общее количество узлов в дереве определяет время выполнения, так как за пределами своих рекурсивных вызовов каждый вызов выполняет работу $O(1)$. Следовательно, время выполнения определяется количеством вызовов.

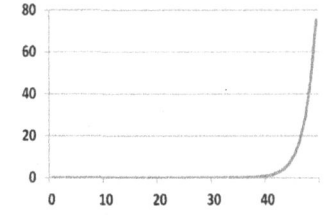
Совет: запомните это обстоятельство для будущих задач. Представление рекурсивных вызовов в виде дерева — отличный способ оценки времени выполнения рекурсивного алгоритма.

Сколько узлов в этом дереве? Пока мы добираемся до базовых случаев (листьев), каждый узел содержит два дочерних узла. Иначе говоря, в каждом узле порождаются две ветви.

Корневой узел имеет два дочерних узла. У каждого из них два своих дочерних узла (таким образом, на уровне «внуков» расположены четыре узла). У каждого из них два своих дочерних узла, и т. д. Если повторить это рассуждение n раз, количество узлов будет приблизительно равно $O(2^n)$.

На самом деле оно немного лучше $O(2^n)$. Взглянув на поддерево, вы можете заметить, что правое поддерево любого узла всегда меньше левого поддерева (кроме листовых узлов и узлов, находящихся непосредственно над ними). Если бы они имели одинаковые размеры, то время выполнения было бы равно $O(2^n)$. Но поскольку размеры правого и левого поддеревьев различны, истинное время выполнения близко к $O(1.6^n)$. Впрочем, запись $O(2^n)$ формально верна, так как она описывает верхнюю границу времени выполнения (см. «O, Θ и Ω» на с. 48). В любом случае, достигается экспоненциальное время выполнения.

В самом деле, если реализовать этот алгоритм на компьютере, вы заметите, что количество секунд возрастает экспоненциально.



Время вычисления N-го числа Фибоначчи в секундах

Нужно поискать возможность оптимизации.

Нисходящее динамическое программирование (или мемоизация)

Проанализируем время рекурсии. Есть ли в дереве идентичные узлы?

Идентичных узлов много. Например, вызов `fib(3)` встречается дважды, а вызов `fib(2)` встречается трижды. Зачем каждый раз вычислять результаты с нуля?

В действительности при вызове `fib(n)` объем работы не должен заметно превышать $O(n)$ вызовов, поскольку существуют всего $O(n)$ возможных значений, которые могут передаваться `fib`. Каждый раз, когда мы вычисляем `fib(i)`, результат следует просто кэшировать и использовать его в будущем.

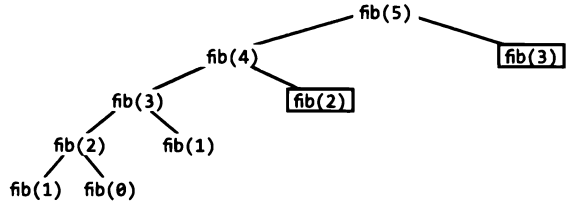
Именно в этом и заключается суть мемоизации.

Всего одно небольшое изменение — и время, необходимое на выполнение этой функции, уменьшится до $O(n)$. Нужно всего лишь кэшировать результаты функции `fibonacci(i)` между вызовами:

```
1  int fibonacci(int n) {
2    return fibonacci(n, new int[n + 1]);
3  }
4
5  int fibonacci(int i, int[] memo) {
6    if (i == 0 || i == 1) return i;
7
8    if (memo[i] == 0) {
9      memo[i] = fibonacci(i - 1, memo) + fibonacci(i - 2, memo);
10   }
11   return memo[i];
12 }
```

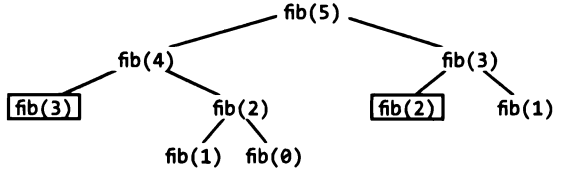
Генерирование 50-го числа Фибоначчи на стандартном компьютере с помощью рекурсивной функции займет около минуты. Метод динамического программирования позволит сгенерировать 10 000-е число Фибоначчи за доли миллисекунды (но не забывайте, что размера типа `int` может оказаться недостаточно).

Если нарисовать дерево рекурсии, оно выглядит примерно так (прямоугольники изображают кэшированные вызовы, которые немедленно возвращают управление):



Сколько узлов теперь содержит дерево? Можно заметить, что дерево теперь уходит в глубину приблизительно n. Каждый из этих узлов также имеет один дочерний узел, так что в результате дерево содержит приблизительно $2n$ дочерних узлов. Таким образом, время выполнения составляет $O(n)$.

Часто бывает полезно представить дерево рекурсии в следующем виде:



На самом деле рекурсия происходит *не так*. Однако с расширением узлов, находящихся на более высоком уровне, вместо нижних узлов создается дерево, которое разрастается в первую очередь в ширину (а не в глубину). Иногда такое представление упрощает вычисление количества узлов в дереве. Здесь мы всего лишь изменяем то, какие узлы раскрываются, а какие возвращают кэшируемые значения. Попробуйте применить этот прием, если у вас возникнут трудности при вычислении времени выполнения задачи динамического программирования.

Восходящее динамическое программирование

Мы также можем взять этот метод и реализовать его средствами восходящего динамического программирования. Считайте, что происходит то же самое, что и при рекурсивном подходе с мемоизацией, но в обратном порядке.

Сначала мы вычисляем `fib(1)` и `fib(0)` — случаи, которые, как нам уже известно, являются базовыми. Затем эти результаты используются для вычисления `fib(2)`. После этого предыдущие ответы используются для вычисления `fib(3)`, `fib(4)` и т. д.

```
1  int fibonacci(int n) {
2    if (n == 0) return 0;
3    else if (n == 1) return 1;
4
5    int[] memo = new int[n];
6    memo[0] = 0;
7    memo[1] = 1;
8    for (int i = 2; i < n; i++) {
9      memo[i] = memo[i - 1] + memo[i - 2];
10    }
11    return memo[n - 1] + memo[n - 2];
12 }
```

Если действительно хорошо поразмыслить над задачей, становится понятно, что значение `memo[i]` используется только для `memo[i+1]` и `memo[i+2]`. После этого хранить его не нужно. Следовательно, можно вообще избавиться от таблицы `memo` и ограничиться хранением нескольких переменных.

```
1  int fibonacci(int n) {
2    if (n == 0) return 0;
3    int a = 0;
4    int b = 1;
5    for (int i = 2; i < n; i++) {
6      int c = a + b;
7      a = b;
8      b = c;
9    }
10    return a + b;
11  }
```

По сути, результаты двух последних значений Фибоначчи сохраняются в переменных `a` и `b`. При каждой итерации вычисляется следующее значение (`c = a + b`), после чего пара (`b, c = a + b`) перемещается в (`a, b`).

Такое объяснение может показаться чрезмерным для столь простой задачи, но настоящее понимание процесса пригодится вам в решении более сложных задач. Решение задач этой главы, многие из которых используют динамическое программирование, поможет укрепить понимание темы.

Дополнительные темы: Доказательство методом индукции (с. 668).

Вопросы собеседования

8.1. Ребенок поднимается по лестнице из *n* ступенек. За один шаг он может переместиться на одну, две или три ступеньки. Реализуйте метод, рассчитывающий количество возможных вариантов перемещения ребенка по лестнице.

Подсказки: 152, 178, 217, 237, 262, 359

8.2. Робот стоит в левом верхнем углу сетки, состоящей из *r* строк и *c* столбцов. Робот может перемещаться в двух направлениях: вправо и вниз, но некоторые ячейки сетки заблокированы, то есть робот через них проходить не может. Разработайте алгоритм построения маршрута от левого верхнего до правого нижнего угла.

Подсказки: 331, 360, 388

8.3. Определим «волшебный» индекс для массива `A[0...n-1]` как индекс, для которого выполняется условие `A[i]=i`. Для заданного отсортированного массива, не содержащего одинаковых значений, напишите метод поиска «волшебного» индекса в массиве A (если он существует).

Дополнительно

Что произойдет, если массив будет содержать одинаковые значения?

Подсказки: 170, 204, 240, 286, 340

8.4. Напишите метод, возвращающий все подмножества заданного множества.

Подсказки: 273, 290, 338, 354, 373

8.5. Напишите рекурсивную функцию для умножения двух положительных целых чисел без использования оператора *. Допускается использование операций сложения, вычитания и поразрядного сдвига, но их количество должно быть минимальным.

Подсказки: 166, 203, 227, 234, 246, 280

8.6. В классической задаче о ханойских башнях имеются 3 башни и N дисков разного размера, которые могут перекладываться между башнями. В начале головоломки диски отсортированы по возрастанию размера сверху вниз (то есть каждый диск лежит на диске большего размера). Установлены следующие ограничения:

(1) За один раз можно переместить только один диск.

(2) Диск кладется на вершину другой башни.

(3) Диск нельзя положить на диск меньшего размера.

Напишите программу для перемещения дисков с первой башни на последнюю с использованием стеков.

Подсказки: 144, 224, 250, 272, 318

8.7. Напишите метод для вычисления всех перестановок строки, состоящей из уникальных символов.

Подсказки: 150, 185, 200, 267, 278, 309, 335, 356

8.8. Напишите метод для вычисления всех перестановок строки, символы которой не обязаны быть уникальными. В списке перестановок дубликатов быть не должно.

Подсказки: 161, 190, 222, 255

8.9. Реализуйте алгоритм для вывода всех корректных (правильно открытых и закрытых) комбинаций из *n* пар круглых скобок.

Пример:

Ввод: 3

Вывод: ((())), (()()), (())(), ()(()), ()()()

Подсказки: 138, 174, 187, 209, 243, 265, 295

8.10. Реализуйте функцию заливки краской, которая поддерживается во многих графических редакторах. Дана плоскость (двумерный массив цветов), точка и цвет, которым нужно заполнить все окружающее пространство, изначально окрашенное в другой цвет.

Подсказки: 364, 382

8.11. Дано неограниченное количество монет достоинством 25, 10, 5 и 1 цент. Напишите код, определяющий количество способов представления *n* центов.

Подсказки: 300, 324, 343, 380, 394

8.12. Напишите алгоритм, находящий все варианты расстановки восьми ферзей на шахматной доске размером 8×8 так, чтобы никакие две фигуры не располагались на одной горизонтали, вертикали или диагонали (учитываются не только две главные, но и все остальные диагонали).

Подсказки: 308, 350, 371

8.13. Имеется штабель из n ящиков шириной w_i, высотой h_i и глубиной d_i. Ящики нельзя поворачивать, добавлять ящики можно только на верх штабеля. Каждый нижний ящик в стопке по высоте, ширине и глубине должен быть строго больше ящика, который находится на нем. Реализуйте метод для вычисления высоты самого высокого штабеля (высота штабеля равна сумме высот всех ящиков).

Подсказки: 155, 194, 214, 260, 322, 368, 378

8.14. Дано логическое выражение, построенное из символов `0`, `1`, `&`, `|` и `^`, и нужное логическое значение `result`. Напишите функцию, подсчитывающую количество вариантов расстановки в логическом выражении круглых скобок, для которых результат выражения равен `result`.

Пример:

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

Подсказки: 148, 168, 197, 305, 327

Дополнительные вопросы: связные списки (2.2, 2.5, 2.6), стеки и очереди (3.3), деревья и графы (4.2, 4.3, 4.4, 4.5, 4.8, 4.10, 4.11, 4.12), головоломки (6.6), сортировка и поиск (10.5, 10.9, 10.10), C++ (12.8), задачи умеренной сложности (16.11), сложные задачи (17.4, 17.6, 17.8, 17.12, 17.13, 17.15, 17.16, 17.24, 17.25).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

9

Масштабируемость и проектирование систем

Несмотря на пугающее название, вопросы о масштабируемости чаще всего оказываются самыми легкими. В них нет никаких ловушек и хитроумных алгоритмов (по крайней мере обычно их не бывает). У большинства людей проблемы возникают из-за того, что им кажется, будто в таких задачах есть нечто особенное — что они требуют каких-то тайных знаний.

На самом деле это не так. Такие вопросы просто разработаны для того, чтобы интервьюер увидел, как вы работаете в реальных условиях. Если ваш начальник прикажет вам спроектировать систему, как вы будете действовать?

Вот почему к решению подобных задач следует подходить именно с этих позиций. Возьмитесь за задачу так, как если бы вы были на работе. Задавайте вопросы. Общайтесь с интервьюером. Обсуждайте достоинства и недостатки.

В этой главе мы рассмотрим некоторые ключевые концепции, но речь идет не о запоминании концепций. Да, хорошее понимание высокоуровневых концепций проектирования систем может пригодиться, но выбор процесса гораздо важнее. Решения бывают хорошими и плохими, идеальных решений не бывает.

Работа с вопросами

- **Общайтесь**: главная цель вопросов по системному проектированию — оценка ваших способностей по общению. Находитесь в контакте с интервьюерами, задавайте им вопросы. Прислушивайтесь к их мнению при обсуждении недостатков вашей системы.
- **Начните с общей картины**: не хватайтесь за конкретную часть алгоритма и не концентрируйтесь на одной части.
- **Используйте доску**: с самого начала подойдите к доске и нарисуйте схему того, что вы предлагаете. Это поможет интервьюеру понять логику предлагаемого решения.
- **Признавайте недостатки**: вероятно, у интервьюера будут критические замечания. Не отмахивайтесь от них. Признайте недостатки, на которые укажет интервьюер, и внесите соответствующие изменения.
- **Будьте осторожны с предположениями**: неправильное предположение способно радикально изменить задачу. Например, если ваша система генерирует аналитику/статистику по набору данных, очень важно, чтобы эта аналитика создавалась на самых актуальных данных.

- **Излагайте свои предположения в явном виде**: если вы делаете предположения, сформулируйте их. Это позволит интервьюеру поправить вас, если предположения будут ошибочными, и покажет, что вы по крайней мере осознаете, какие предположения делаете.
- **Предоставляйте оценки там, где это необходимо**: во многих случаях необходимые данные могут отсутствовать. Например, если вы проектируете веб-бота, вам может понадобиться оценка пространства, необходимого для хранения всех URL. Оценка может быть сформирована на базе других имеющихся данных.
- **Направляйте ход обсуждения**: кандидат должен находиться «за рулем». Это не означает, что вы не должны общаться с интервьюером; более того, общаться с ним необходимо. Тем не менее в обсуждении вы должны играть активную роль. Задавайте вопросы. Будьте готовы к компромиссным решениям. Продолжайте исследовать задачу «в глубину». Не останавливайтесь на достигнутом и вносите усовершенствования.

Все эти рекомендации относятся в основном к процессу, а не к конечному результату.

Проектирование: шаг за шагом

Представьте, что вам предложено спроектировать систему: скажем, аналог TinyURL. Как вы будете действовать? Скажете: «Хорошо» и закроетесь в кабинете, чтобы спроектировать систему самостоятельно? Вряд ли. Скорее всего, сначала вы захотите задать множество вопросов. Именно так следует действовать во время собеседования.

Шаг 1. Определите масштаб задачи

Нельзя спроектировать систему, если вы не знаете, что проектируете. Определение масштаба задачи — важная часть процесса; вы должны быть уверены в том, что строите именно то, что нужно интервьюеру. К тому же может оказаться, что именно этот навык хочет оценить интервьюер.

Если вам предлагается «спроектировать аналог TinyURL», сначала необходимо точно понять, что именно нужно реализовать. Смогут ли пользователи самостоятельно задать короткий URL-адрес? А может, адреса будут генерироваться автоматически? Нужно ли вести статистику по щелчкам? Должны ли URL-адреса оставаться навечно или их срок жизни ограничен?

Вы должны получить ответы на эти вопросы, прежде чем идти дальше.

Составьте список вопросов, а также перечислите основные возможности или варианты использования. Например, для TinyURL этот список может выглядеть так:

- Сокращение URL-адреса до TinyURL.
- Аналитика по URL.
- Получение URL-адреса, связанного с TinyURL.
- Управление учетными записями пользователей и ссылками.

Шаг 2. Делайте разумные предположения

Никто не запрещает вам делать предположения (когда это необходимо), но предположения должны быть разумными. Например, было бы неразумно предположить, что система будет обслуживать 100 пользователей в день или что в вашем распоряжении неограниченный объем памяти.

С другой стороны, будет разумно проектировать систему с расчетом на максимум 1 000 000 новых URL в день. Такое предположение поможет вам вычислить объем данных, которые должна сохранять система.

Некоторые предположения могут требовать «ощущения продукта» (и в этом нет ничего плохого). Например, можно ли предположить, что данные остаются актуальными в течение 10 минут? Зависит от обстоятельств. Скажем, только что введенный URL-адрес должен активизироваться немедленно, тогда как устаревание статистики на 10 минут может быть вполне допустимо. Обсудите с интервьюером предположения такого рода.

Шаг 3. Нарисуйте главные компоненты

Встаньте и подойдите к доске. Нарисуйте диаграмму с основными компонентами системы. Возможно, на ней будет присутствовать внешний сервер (или группа серверов), который будет получать данные из подсистемы баз данных. Также на схеме может быть другая группа серверов, которые ищут данные в Интернете, и еще одна группа для обработки аналитики. Нарисуйте схему со своим видением системы.

Пройдите по своей системе от одной конечной точки до другой, чтобы понять суть потока операций. Пользователь ввел новый URL-адрес. Что происходит после этого?

Иногда в этой фазе бывает полезно игнорировать серьезные проблемы масштабируемости и притвориться, что будет достаточно простых, очевидных решений. Серьезными проблемами вы займетесь на шаге 4.

Шаг 4. Выявление ключевых проблем

После формирования базовой архитектуры сосредоточьтесь на ключевых проблемах. Какие «узкие места» и общие проблемы присутствуют в системе?

Например, если вы проектируете аналог TinyURL, следует рассмотреть возможность того, что обращения к одним URL-адресам будут относительно редкими, а на других будут внезапно возникать пиковые нагрузки, скажем, из-за публикации URL на Reddit или другом популярном форуме. Иногда слишком частых обращений к базе удается избежать.

Возможно, какую-то полезную информацию может предоставить интервьюер. В таком случае воспользуйтесь его рекомендациями.

Шаг 5. Изменение проекта с учетом ключевых проблем

После выявления ключевых проблем следует адаптировать к ним проект. Для этого может потребоваться как серьезная переработка, так и второстепенные изменения (например, введение кэширования).

Оставайтесь у доски и обновляйте диаграмму по мере изменения решения.

Не скрывайте ограничения вашего решения. Скорее всего, интервьюеру о них все равно известно, поэтому важно показать, что вы эти ограничения тоже понимаете.

Масштабируемые алгоритмы: шаг за шагом

Иногда перед вами не ставят задачу спроектировать целую систему. Вам предлагают спроектировать отдельную функциональную возможность или алгоритм, но так, чтобы обеспечить их масштабируемость. а может быть, в алгоритме присутствует одна часть, которая является «настоящей» темой более широкого вопроса из области проектирования.

В таких случаях попробуйте применить следующий метод.

Шаг 1. Задайте вопросы

Как и в предыдущем процессе, задайте вопросы, чтобы убедиться в том, что вы правильно поняли задание. Возможно, интервьюер пропустил какие-то подробности (случайно или намеренно). Невозможно решить задачу, если вы недостаточно четко понимаете ее.

Шаг 2. Абстрагируйтесь от реальности

Представьте, что все данные хранятся на одном компьютере и не существует никаких ограничений памяти. Как бы вы решили задачу? Ответ на этот вопрос позволит построить общую схему решения.

Шаг 3. Вернитесь к реальности

Вернитесь к исходной задаче. Сколько данных можно разместить в одном компьютере? Какие проблемы могут возникнуть при распределении данных? Типичные задачи такого рода — определение того, как организовать логическое разбиение данных и как отдельный компьютер должен понять, куда обращаться за недостающей информацией.

Шаг 4. Решите проблемы

Подумайте о том, как избавиться от проблем, обнаруженных на шаге 2. Помните, что решение может заключаться как в полном устранении проблемы, так и в простом снижении ее серьезности. Обычно удается продолжить использование схемы, построенной на шаге 1 (возможно, с небольшими модификациями), но иногда приходится вносить более масштабные изменения.

Используйте итеративный подход. Как только проблемы, выявленные на шаге 2 , будут устранены, могут появиться новые, и вам придется заняться ими.

Ваша цель — не спроектировать сложную систему, на которую компании придется потратить миллионы долларов, а продемонстрировать, что вы можете анализировать и решать задачи подобного плана. В этом отношении нет ничего лучше поиска недостатков в ваших собственных решениях.

Ключевые концепции

Хотя вопросы по проектированию систем проверяют не столько знания, сколько навыки, некоторые концепции существенно упростят вашу задачу. Ниже приведена краткая сводка. Каждый из ее пунктов является глубокой, сложной темой, поэтому за дополнительной информацией стоит обратиться к интернет-ресурсам.

Горизонтальное и вертикальное масштабирование

Система может масштабироваться одним из двух способов.

- Под *вертикальным масштабированием* понимается наращивание ресурсов отдельного узла системы. Например, на сервере можно установить дополнительную память, чтобы он лучше справлялся с изменениями нагрузки.
- Под *горизонтальным масштабированием* понимается увеличение количества узлов. Например, в системе устанавливаются дополнительные серверы, что приводит к снижению нагрузки на отдельный сервер.

Вертикальное масштабирование обычно проще горизонтального, но его возможности ограниченны. Возможности наращивания памяти или дискового пространства не бесконечны.

Распределение нагрузки

Некоторые внешние части масштабируемого сайта обычно размещаются за распределителем нагрузки. Это позволяет системе равномерно распределять нагрузку, чтобы сбой одного сервера не приводил к отказу всей системы. Конечно, для этого необходимо построить сеть из клонированных серверов, которые выполняют фактически один и тот же код и имеют доступ к одним и тем же данным.

Денормализация баз данных и NoSQL

Соединения (joins) в реляционных базах данных сильно замедляются с ростом системы. По этой причине их обычно следует избегать.

Одной из составляющих этой политики является *денормализация* — хранение избыточной информации в базе данных для ускорения чтения. Представьте базу данных с описаниями проектов и задач (один проект может состоять из нескольких задач). Допустим, вы хотите получить информацию о задаче, включая название проекта. Вместо того, чтобы выполнять соединение между таблицами, можно хранить название проекта в таблице задач (а не только в таблице проектов).

Также можно воспользоваться базой данных NoSQL. Базы данных NoSQL не поддерживают соединения, а данные в них имеют другую структуру. Они спроектированы с расчетом на эффективное масштабирование.

Сегментация баз данных

Под *сегментацией* (sharding) понимается схема разбиения данных по нескольким машинам, при которой сохраняется возможность определить, на какой машине находятся те или иные данные.

Некоторые стандартные способы сегментации:

- **Вертикальная сегментация**: фактически это сегментация по функциональности. Например, при построении социальной сети можно выделить один сегмент для таблиц, относящихся к профилям, другой для сообщений и т. д. Недостаток такого способа заключается в том, что если одна из таблиц станет очень большой, может возникнуть необходимость в повторном сегментировании базы данных (возможно, с использованием другой схемы разбиения).
- **Сегментация по ключу (или по результату хеширования)**: некоторая часть данных (например, идентификатор) используется для разбиения. Очень простая схема — выделение N серверов и размещение данных на сервере `mod(key,n)`. К недостаткам этого метода следует отнести то, что он практически означает жестко фиксированное количество серверов. Установка дополнительных серверов потребует перераспределения всех данных — это обойдется очень дорого.
- **Сегментация по каталогу**: в системе создается таблица с информацией о местонахождении данных. При такой схеме установка дополнительных серверов происходит относительно просто, но у нее есть и два серьезных недостатка. Во-первых, таблица каталога может стать единой точкой отказа («критическим звеном») системы. Во-вторых, постоянные обращения к таблице отражаются на быстродействии.

Во многих архитектурах применяются комбинированные решения с несколькими схемами сегментации.

Кэширование

Кэширование способно заметно ускорить получение результатов. Кэш представляет собой простое хранилище пар «ключ—значение», которое обычно располагается между уровнем приложения и хранилищем данных.

Когда приложение запрашивает некоторую информацию, оно сначала проверяет содержимое кэша. Если в кэше соответствующий ключ отсутствует, приложение обращается за данными к основному хранилищу.

В вашем решении может использоваться прямое кэширование запроса и его результатов. Также возможен и другой вариант — кэширование конкретного объекта (например, сгенерированной части веб-сайта или списка последних сообщений в блоге).

Асинхронная обработка и очереди

В идеале медленные операции должны выполняться асинхронно, чтобы пользователю не приходилось подолгу дожидаться завершения процесса.

В некоторых случаях такие операции могут выполняться с опережением. Например, вы можете сформировать очередь заданий на обновление некоторой части веб-сайта. Скажем, в реализации форума одно из таких заданий может заранее генерировать страницу со списком самых популярных публикаций и несколькими комментариями. Такой список может слегка потерять актуальность, но скорее всего, это нормально — по крайней мере лучше, чем заставлять пользователя дожидаться загрузки сайта просто потому, что кто-то добавил новый комментарий и кэшированная версия страницы стала недействительной.

В других случаях можно попросить пользователя подождать и оповестить его о завершении операции. Наверняка вам уже встречалась эта функция на сайтах: вы активизируете некоторую часть функциональности сайта и получаете сообщение о том, что на импортирование данных потребуется несколько минут, но когда это будет сделано, вы получите оповещение.

Сетевые метрики

Важнейшие метрики, относящиеся к передаче данных по сети:

- **Пропускная способность**: максимальный объем данных, которые могут быть переданы за единицу времени. Обычно выражается в битах в секунду (или аналогичным образом — например, в гигабайтах в секунду).
- **Скорость передачи**: если пропускная способность определяет максимальный объем данных, которые *могут быть* переданы за единицу времени, при вычислении скорости передачи используется фактический объем переданных данных.
- **Задержка**: время, необходимое для перемещения данных из одной точки в другую. Иначе говоря, это промежуток времени между отправкой информации (обычно очень маленького пакета данных) и ее получением.

Представьте, что у вас имеется конвейер, который перемещает детали по фабрике. Задержка соответствует времени, необходимому для перемещения деталей с одного конца ленты на другой, а скорость передачи — количеству деталей, перемещаемых конвейером в секунду.

- Расширение ленты конвейера не изменит задержку. С другой стороны, оно изменит пропускную способность и скорость передачи. На конвейере будет помещаться большее количество деталей, что позволит перемещать больше заготовок за единицу времени.
- Сокращение длины конвейера уменьшит задержку, так как детали будут проводить меньше времени в движении. Пропускная способность и скорость передачи при этом останутся неизменными: за единицу времени по конвейеру будет перемещаться то же количество деталей.
- Ускорение движения конвейера приведет к изменению всех трех характеристик. Время, необходимое для перемещения по фабрике, сократится. Кроме того, за единицу времени будет перемещаться большее количество деталей.
- Пропускная способность определяет количество деталей, которые могут перемещаться за единицу времени при идеальных условиях. Скорость передачи определяет их фактическое количество — например, если машины работают с перебоями.

О задержке легко забыть, но она бывает очень важна в конкретных ситуациях. Например, в сетевых играх задержка может быть критичной. Как играть по сети в типичную спортивную игру (скажем, в футбол для двоих игроков), если вы не будете мгновенно получать информацию о перемещениях противника? Кроме того, в отличие от скорости передачи, которую иногда можно повысить за счет сжатия данных, возможностей для улучшения задержки обычно немного.

MapReduce

Технология MapReduce часто ассоциируется с Google, но на самом деле она находит более широкое применение. Программа MapReduce обычно используется для обработки больших объемов данных.

Как подсказывает само название, для работы MapReduce требуется запрограммировать две операции: отображение (`Map`) и свертку (`Reduce`). Все остальное делает сама система.

- Операция `Map` получает данные и выдает пару *<ключ,значение>*.
- Операция `Reduce` получает ключ и множество связанных значений и каким-то образом обрабатывает их, создавая новый ключ и значение. Результаты могут быть снова переданы программе `Map` для дальнейшей свертки.

MapReduce позволяет выполнять большое количество вычислений параллельно, что улучшает масштабируемость обработки больших объемов данных.

За дополнительной информацией обращайтесь к разделу «MapReduce», с. 680.

Дополнительные факторы

Кроме перечисленных выше концепций, при проектировании системы также нужно учесть следующие факторы.

- **Отказы**: практически в любой части системы может произойти отказ. Вы должны запланировать многие (если не все) из потенциальных отказов.
- **Доступность и надежность**: доступность — функция процента времени, в течение которого система находится в работоспособном состоянии; надежность — функция вероятности того, что система остается работоспособной в течение некоторого периода времени.
- **Интенсивное чтение и интенсивная запись**: выполнение большого объема чтения или записи данных влияет на архитектуру системы. В системах с интенсивной записью следует рассмотреть возможность формирования очереди записи (но рассмотрите возможность отказа!). В системах с интенсивным чтением стоит организовать кэширование. Также при этом могут измениться и другие решения, связанные с проектированием.
- **Безопасность**: угрозы безопасности способны обернуться катастрофой для системы. Подумайте, с какими угрозами может столкнуться система, и обойдите их на уровне архитектуры.

Этот список дает только первое представление о возможных проблемах в системе. Помните, что в ходе собеседования следует открыто говорить о принимаемых компромиссных решениях.

Идеальных систем не бывает

Не существует единственно верной архитектуры для TinyURL, Google Maps или любой другой системы, которая работала бы идеально (хотя существует великое множество архитектур, которые работают ужасно). Всегда существуют плюсы

и минусы. Два специалиста могут предложить совершенно разные решения, и оба будут отличными при разных допущениях.

В таких задачах вы должны продемонстрировать свое умение понять варианты использования, определить масштаб задачи, сделать разумные допущения, построить добротное решение на базе этих допущений и не скрывать слабые стороны вашего решения. Никто не ждет от вас идеального результата.

Пример: найдите все документы, содержащие список слов

Имеется миллион документов. Как бы вы организовали поиск всех документов, содержащих слова из определенного списка? Слова могут располагаться в тексте в произвольном порядке, но они не могут быть фрагментами других слов. То есть если мы ищем `book`, то `bookkeeper` не удовлетворяет критериям поиска.

Сначала следует разобраться, является операция одноразовой или же процедура `findWords` будет выполняться многократно. Давайте предположим, что `findWords` будет многократно использоваться для одного и того же набора документов, а следовательно, дополнительные затраты на предварительную обработку можно считать оправданными.

Шаг 1

На первом шаге нужно забыть о миллионе документов и сделать вид, что нужно обработать всего дюжину файлов. Как бы вы реализовали `findWords` в этом случае? (Подсказка: перед тем, как читать дальше, попытайтесь решить эту задачу самостоятельно.)

Один из способов — предварительно обработать каждый документ и построить индекс в формате хеш-таблицы. Эта таблица связывает каждое слово со списком документов, содержащих это слово.

```
"books" -> {doc2, doc3, doc6, doc8}
"many" -> {doc1, doc3, doc7, doc8, doc9}
```

Чтобы найти документы, содержащие слова `many` и `books`, достаточно обратиться к таблице и выбрать пересечение этих множеств — `{doc3, doc8}`.

Шаг 2

Вернемся к исходной задаче. Какие проблемы могут возникнуть, если число документов увеличивается до миллиона? Скорее всего, документы будут храниться на нескольких компьютерах. Более того, при определенных условиях — большое количество слов и их повторяемость в документах — полная хеш-таблица может не уместиться на одном компьютере. Будем считать, что дело обстоит именно так.

Основные проблемы в такой постановке задачи:

1. Как организовать распределение хеш-таблицы? Можно разбить ее по ключевым словам — например, расположить полный список документов, содержащих заданное слово, на одной машине. Или мы можем использовать другой

принцип — на компьютере будет находиться хеш-таблица только для заданного подмножества документов.

2. Когда мы решим, каким образом разделить данные, может потребоваться обработать документ, находящийся на одной машине, и передать результаты на другую. Как работает этот процесс? (Если хеш-таблица разделена по документам, то этот шаг не нужен.)
3. Необходимо придумать метод для получения информации о том, какие данные и на каком компьютере будут храниться. Как выглядит таблица поиска и где она хранится?

Это всего лишь три проблемы, могут быть и другие.

Шаг 3

На третьем шаге мы ищем решения каждой из перечисленных задач. Одно из возможных решений — упорядочить слова по алфавиту, чтобы каждый компьютер обслуживал определенный диапазон слов (например, от `after` до `apple`).

Можно реализовать простой алгоритм перебора всех ключевых слов в алфавитном порядке и сохранения максимально возможного объема данных на каждом компьютере. Когда ресурсы одного компьютера будут исчерпаны, мы переходим к следующему компьютеру.

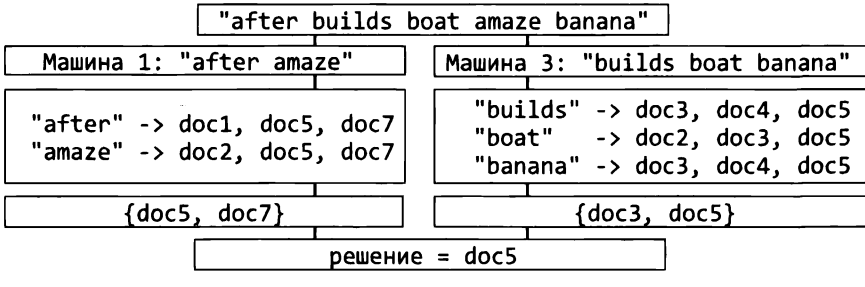
Преимущество этого подхода — простая и небольшая таблица поиска (так как в ней хранится только диапазон значений), при этом копии таблицы поиска могут сохраняться на каждой машине. Недостаток заключается в том, что при добавлении новых документов придется проводить сдвиг ключевых слов, который обойдется очень дорого.

Чтобы найти все документы, содержащие строки из списка, можно отсортировать список и передать каждому компьютеру поисковый запрос по словам, которые хранятся на конкретном компьютере. Если нам нужно найти `after builds boat amaze banana`, машина № 1 получит запрос `{"after", "amaze"}`.

Машина № 1 найдет документы, содержащие слова `after` и `amaze`, а затем найдет пересечение списков документов. Машина № 3 проделает такую же операцию для слов `{"banana", "boat", "builds"}`.

На заключительном этапе исходный компьютер найдет пересечение результатов, полученных от компьютеров № 1 и № 3.

Следующая схема объясняет этот процесс.



Вопросы собеседования

Эти вопросы имитируют обстановку реального собеседования, поэтому они не всегда четко сформулированы. Подумайте, какие вопросы вы бы задали своему интервьюеру, и сделайте разумные предположения. Возможно, ваши предположения будут сильно отличаться от наших, и ваше решение получится совсем другим. И это нормально!

9.1. Представьте, что вы создаете службу, которая получает и обрабатывает простейшую информацию от 1000 клиентских приложений о курсе акций в конце торгового дня (открытие, закрытие, максимум, минимум). Предположим, что все данные получены и вы можете сами выбрать формат для их хранения. Как должна выглядеть служба, предоставляющая информацию клиентским приложениям? Вы должны обеспечить развертывание, продвижение, мониторинг и обслуживание системы. Опишите различные варианты построения службы, которые вы рассмотрели, и обоснуйте свой подход. Вы можете использовать любые технологии и любой механизм предоставления информации клиентским приложениям.

Подсказки: 385, 396

9.2. Какие структуры данных вы бы стали использовать в очень больших социальных сетях вроде Facebook или LinkedIn? Опишите, как вы будете разрабатывать алгоритм для поиска кратчайшей цепочки знакомств между двумя людьми (Я → Боб → Сьюзи → Джейсон → Ты).

Подсказки: 270, 285, 304, 321

9.3. Представьте, что вы разрабатываете поискового робота. Как избежать зацикливаний при поиске?

Подсказки: 334, 353, 365

9.4. Имеются 10 миллиардов URL-адресов. Как обнаружить все дублирующиеся документы? Дубликатом в данном случае считается совпадение URL-адресов.

Подсказки: 326, 347

9.5. Представьте веб-сервер упрощенной поисковой системы. В системе есть 100 компьютеров, обрабатывающих поисковые запросы, которые могут генерировать запрос `processSearch(string query)` к другому кластеру компьютеров для получения результата. Компьютер, отвечающий на запрос, выбирается случайным образом; нет гарантий, что один запрос всегда будет передаваться одной и той же машине. Метод `processSearch` очень затратный. Разработайте механизм кэширования новых запросов. Объясните, как будет обновляться кэш при изменении данных.

Подсказки: 259, 274, 293, 311

9.6. Крупный интернет-магазин хочет составить список самых популярных продуктов — в целом и по отдельным категориям. Например, один продукт может стоять на 1056-м месте в общем списке популярности, но при этом занимать 13-е место в категории «Спортивное оборудование», и 24-е место в категории «Безопасность». Опишите, как бы вы подошли к проектированию такой системы.

Подсказки: 142, 158, 176, 189, 208, 223, 236, 244

9.7. Объясните, как бы вы спроектировали сервис персонального финансового менеджера (аналог Mint.com). Система должна подключаться к банковским счетам, анализировать характер расходов и давать рекомендации.

Подсказки: 162, 180, 199, 212, 247, 276

9.8. Спроектируйте систему — аналог Pastebin, в которой пользователь может ввести фрагмент текста и получить случайно сгенерированный URL-адрес для обращения к нему.

Подсказки: 165, 184, 206, 232

Дополнительные вопросы: объектно-ориентированное проектирование (7.7).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

10

Сортировка и поиск

Понимание общих принципов поиска и сортировки очень важно, так как большинство задач поиска и сортировки являются небольшими модификациями известных алгоритмов. Самый правильный подход — освоить известные алгоритмы сортировки и разобраться, для каких ситуаций подходит каждый из них.

Допустим, вам предложено следующее задание: провести сортировку очень большого массива `Person`, упорядочив его элементы по возрасту человека (в порядке возрастания).

Нам известно два факта:

- Массив имеет большие размеры, значит, эффективность алгоритма очень важна.
- Сортировка выполняется по возрасту, то есть значения имеют ограниченный диапазон.

Рассмотрев различные алгоритмы, можно прийти к выводу, что для данной ситуации идеально подходит блочная сортировка (bucket sort), также называемая *поразрядной* (radix) сортировкой. Если блоки будут достаточно маленькими (например, 1 год), то время выполнения составит $O(n)$.

Распространенные алгоритмы сортировки

Изучение (или повторение) алгоритмов сортировки — отличный способ повысить свои шансы на собеседовании. Из приведенных ниже алгоритмов на собеседованиях чаще всего встречаются сортировка слиянием, быстрая сортировка и блочная сортировка.

Пузырьковая сортировка | Время выполнения в худшем и среднем случае: $O(n^2)$. Затраты памяти: $O(1)$.

Обработка начинается с первых элементов массива; они меняются местами, если первое значение больше второго. Затем алгоритм переходит к следующей паре и так далее, пока массив не будет отсортирован. В процессе сортировки меньшие элементы постепенно «всплывают» в начало списка.

Сортировка выбором | Время выполнения в худшем и среднем случае: $O(n^2)$. Затраты памяти: $O(1)$.

Сортировка выбором — алгоритм для новичков: просто, но неэффективно. С помощью линейного сканирования ищется наименьший элемент, а затем меняется местами с первым элементом. Затем с помощью линейного сканирования ищется второй наименьший элемент и снова перемещается в начало. Алгоритм продолжает работу, пока массив не будет полностью отсортирован.

Сортировка слиянием | Время выполнения в худшем и среднем случае: $O(n \log(n))$. Память: зависит от задачи.

При сортировке слиянием массив делится пополам, каждая половина сортируется по отдельности, и затем делается обратное объединение. К каждой из половин применяется один и тот же алгоритм сортировки. Рекурсия завершается объединением двух массивов, состоящих из одного элемента.

Метод слияния копирует все элементы из целевого массива во вспомогательные, разделяя левую и правую половины (`helperLeft` и `helperRight`). Затем алгоритм перебирает массив `helper`, копируя наименьший элемент каждой половины в массив. В итоге мы копируем оставшиеся элементы в целевой массив.

```
1  void mergesort(int[] array) {
2    int[] helper = new int[array.length];
3    mergesort(array, helper, 0, array.length - 1);
4  }
5
6  void mergesort(int[] array, int[] helper, int low, int high) {
7    if (low < high) {
8      int middle = (low + high) / 2;
9      mergesort(array, helper, low, middle);     // Сортировка левой половины
10     mergesort(array, helper, middle+1, high); // Сортировка правой половины
11     merge(array, helper, low, middle, high);  // Слияние
12   }
13 }
14
15 void merge(int[] array, int[] helper, int low, int middle, int high) {
16   /* Обе половины копируются в массив helper */
17   for (int i = low; i <= high; i++) {
18     helper[i] = array[i];
19   }
20
21   int helperLeft = low;
22   int helperRight = middle + 1;
23   int current = low;
24
25   /* Перебор массива helper. Сравниваем левую и правую половины
26    * и копируем меньший элемент из двух половин в исходный массив. */
27   while (helperLeft <= middle && helperRight <= high) {
28     if (helper[helperLeft] <= helper[helperRight]) {
29       array[current] = helper[helperLeft];
30       helperLeft++;
31     } else { // Если правый элемент меньше левого
32       array[current] = helper[helperRight];
33       helperRight++;
34    }
35     current++;
36   }
37
38   /* Остаток левой стороны массива копируется в целевой массив */
39   int remaining = middle - helperLeft;
40   for (int i = 0; i <= remaining; i++) {
41     array[current + i] = helper[helperLeft + i];
42   }
43 }
```

Обратите внимание, что в целевой массив копируются только оставшиеся элементы левой половины вспомогательного массива. Почему не правой половины? Правая половина и не должна копироваться, потому что она *уже* там находится.

Рассмотрим для примера массив [1, 4, 5 || 2, 8, 9] (знак || указывает точку разбиения). До момента слияния половин оба массива — вспомогательный и целевой — заканчиваются элементами [8, 9]. Когда мы копируем в целевой массив больше четырех элементов (1, 4, 5, и 2), элементы [8, 9] продолжают оставаться в обоих массивах. Нет никакой необходимости копировать их.

Пространственная сложность сортировки слиянием равна $O(n)$ из-за дополнительного пространства, необходимого для слияния частей массива.

Быстрая сортировка | Время выполнения в среднем случае: $O(n \log(n))$, в худшем случае: $O(n^2)$. Затраты памяти: $O(\log(n))$.

При быстрой сортировке из массива выбирается случайный (опорный) элемент, и все элементы массива располагаются по принципу: меньшие–равные–большие относительно выбранного элемента. Такую декомпозицию эффективнее всего осуществлять серией перестановок (см. далее).

Если многократно делить массив (и его подмассивы) относительно случайного элемента, то в результате получим отсортированный массив. Поскольку нет никаких гарантий, что опорный элемент является медианой (даже приближенно), наша сортировка окажется очень медленной — в худшем случае $O(n^2)$.

```
1  void quickSort(int[] arr, int left, int right) {
2    int index = partition(arr, left, right);
3    if (left < index - 1) { // Сортировка левой половины
4      quickSort(arr, left, index - 1);
5    }
6    if (index < right) { // Сортировка правой половины
7      quickSort(arr, index, right);
8    }
9  }
10
11  int partition(int[] arr, int left, int right) {
12    int pivot = arr[(left + right) / 2]; // Выбор центральной точки
13    while (left <= right) {
14      // Найти слева элемент, который должен быть справа
15      while (arr[left] < pivot) left++;
16
17      // Найти справа элемент, который должен быть слева
18      while (arr[right] > pivot) right--;
19
20      // Поменять элементы местами и сместить индексы left и right
21      if (left <= right) {
22        swap(arr, left, right); // Элементы меняются местами
23        left++;
24        right--;
25      }
26    }
27    return left;
28  }
```

Поразрядная сортировка | Время выполнения: $O(kn)$ (см. ниже).

Поразрядная сортировка — это алгоритм сортировки целых (и некоторых других) чисел, использующий факт, что целые числа представляются конечным числом битов. Мы группируем числа по каждому разряду. Например, массив целых чисел сначала сортируется по первому разряду (чтобы все 0 собрались в группу). Затем каждая из групп сортируется по следующему разряду. Процесс повторяется, пока весь массив не будет отсортирован.

В отличие от алгоритмов сортировки на базе сравнений, которые в большинстве случаев не могут выполняться быстрее, чем за $O(n \log(n))$, данный алгоритм в худшем случае обеспечивает время выполнения $O(kn)$, где n — количество элементов в массиве, а k — количество проходов алгоритма сортировки.

Алгоритмы поиска

Когда речь заходит об алгоритмах поиска, обычно подразумевается прежде всего бинарный поиск. Действительно, это очень полезный алгоритм.

При бинарном поиске для нахождения элемента x в отсортированном массиве значение x сначала сравнивается со значением из середины массива. Если x меньше средней точки, то алгоритм продолжает поиск в левой половине массива. Если же x больше средней точки, то поиск продолжается в правой половине. Процесс повторяется, а левая и правая половины рассматриваются как подмассивы: x снова сравнивается с элементом из середины подмассива, после чего поиск продолжается в левой или правой части. Все это происходит до тех пор, пока элемент x не будет найден или размер подмассива не сократится до 0.

Заметьте, что хотя концепция довольно проста, не упустить ни одной мелочи сложнее, чем кажется на первый взгляд. Изучите приведенный ниже код, обращая особое внимание на прибавляемые и вычитаемые единицы.

```
1  int binarySearch(int[] a, int x) {
2    int low = 0;
3    int high = a.length - 1;
4    int mid;
5
6    while (low <= high) {
7      mid = (low + high) / 2;
8      if (a[mid] < x) {
9        low = mid + 1;
10     } else if (a[mid] > x) {
11       high = mid - 1;
12     } else {
13       return mid;
14     }
15   }
16   return -1; // Ошибка
17 }
18
19 int binarySearchRecursive(int[] a, int x, int low, int high) {
20   if (low > high) return -1; // Ошибка
```

```
21
22    int mid = (low + high) / 2;
23    if (a[mid] < x) {
24      return binarySearchRecursive(a, x, mid + 1, high);
25    } else if (a[mid] > x) {
26      return binarySearchRecursive(a, x, low, mid - 1);
27    } else {
28      return mid;
29    }
30  }
```

Многочисленные разновидности поиска структур данных не ограничиваются вариациями бинарного поиска; обязательно познакомьтесь с ними поближе. Например, для поиска узлов можно воспользоваться бинарным деревом или хеш-таблицей. Не ограничивайте свои возможности!

Вопросы собеседования

10.1. Имеются два отсортированных массива A и B. В конце массива A существует свободное место, достаточное для размещения массива B. Напишите метод слияния массивов B и A, сохраняющий сортировку.

Подсказки: 332

10.2. Напишите метод сортировки массива строк, при котором анаграммы группируются друг за другом.

Подсказки: 177, 182, 263, 342

10.3. Имеется отсортированный массив из `n` целых чисел, который был циклически сдвинут неизвестное количество раз. Напишите код для поиска элемента в массиве. Предполагается, что массив изначально был отсортирован по возрастанию.

Пример:

Ввод: найти 5 в `{15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}`
Вывод: 8 (индекс числа 5 в массиве)

Подсказки: 298, 310

10.4. Имеется структура данных `Listy` — аналог массива, в котором отсутствует метод определения размера. При этом поддерживается метод `elementAt(i)`, возвращающий элемент с индексом *i* за время $O(1)$. Если значение *i* выходит за границу структуры данных, метод возвращает –1. (По этой причине в структуре данных могут храниться только положительные целые числа). Для заданного экземпляра `Listy`, содержащего отсортированные положительные числа, найдите индекс элемента с заданным значением `x`. Если `x` входит в структуру данных многократно, можно вернуть любой индекс.

Подсказки: 320, 337, 348

10.5. Имеется отсортированный массив строк, в котором могут присутствовать пустые строки. Напишите метод для определения позиции заданной строки.

Пример:

Ввод: строка `"ball"` в массиве `{"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}`

Вывод: 4

Подсказки: 256

10.6. Имеется файл размером 20 Гбайт, состоящий из строк. Как бы вы выполнили сортировку такого файла?

Подсказки: 207

10.7. Имеется входной файл с четырьмя миллиардами неотрицательных целых чисел. Предложите алгоритм для генерирования целого числа, отсутствующего в файле. Считайте, что для выполнения операции доступен 1 Гбайт памяти.

Дополнительно:

А если доступно всего 10 Мбайт памяти? Предполагается, что все значения различны, а количество неотрицательных целых чисел в этом случае не превышает миллиарда.

Подсказки: 235, 254, 281

10.8. Имеется массив с числами от 1 до *N*, где *N* не превышает 32 000. Массив может содержать дубликаты, а значение *N* заранее неизвестно. Как вывести все дубликаты из массива, если доступно всего 4 Кбайт памяти?

Подсказки: 289, 315

10.9. Для заданной матрицы M×N, в которой каждая строка и столбец отсортированы по возрастанию, напишите метод поиска элемента.

Подсказки: 193, 211, 229, 251, 266, 279, 288, 291, 303, 317, 330

10.10. Вы обрабатываете поток целых чисел. Периодически вам нужно находить ранг числа `x` (количество значений `<= x`). Какие структуры данных и алгоритмы необходимы для поддержки этих операций? Реализуйте метод `track(int x)`, вызываемый при генерировании каждого символа, и метод `getRankOfNumber(int x)`, возвращающий количество значений `<=x` (не включая `x`).

Пример:

```
Поток (в порядке появления): 5, 1, 4, 4, 5, 9, 7, 13, 3
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

Подсказки: 301, 376, 392

10.11. Имеется массив целых чисел. Будем называть «пиком» элемент, больший соседних элементов либо равный им, а «впадиной» — элемент, меньший

соседних элементов либо равный им. Например, в массиве {5, 8, 6, 2, 3, 4, 6} элементы {8, 6} являются пиками, а элементы {5, 2} — впадинами. Отсортируйте заданный массив целых чисел в чередующуюся последовательность пиков и впадин.

Пример:

Ввод: `{5, 3, 1, 2, 3}`
Вывод: `{5, 1, 3, 2, 3}`

Подсказки: 196, 219, 231, 253, 277, 292, 316

Дополнительные вопросы: массивы и строки (1.2), рекурсия (8.3), задачи умеренной сложности (16.10, 16.16, 16.21, 16.24), сложные задачи (17.11, 17.26).

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

11

Тестирование

Если вы уже собрались пролистать эту главу со словами «я разработчик, а не тестер», остановитесь и подумайте. Тестирование — очень важная задача для разработчика программного обеспечения, и поэтому вопросы по тестированию могут быть заданы и на вашем собеседовании. Если же вы претендуете на должность тестера, тем более стоит обратить внимание на эту главу.

Задачи тестирования принято делить на четыре категории: 1) тестирование объектов реального мира (например, ручка); 2) тестирование программного продукта; 3) написание тестового кода для функции; 4) исправление существующей ошибки. Мы рассмотрим задачи всех четырех типов.

Помните, что все четыре типа задач предполагают ошибки ввода и «неидеальное» поведение пользователя. Приготовьтесь к некорректному использованию и стройте планы соответственно.

Чего ожидает интервьюер

На первый взгляд кажется, что в заданиях по тестированию от вас всего лишь требуется написать большой список тестов. До некоторой степени это верно — содержательный список тестов действительно понадобится.

Но кроме этого интервьюеры хотят увидеть:

- **Понимание всей картины**: действительно ли вы тот человек, который понимает, что представляет собой данный программный продукт? Можете ли вы правильно назначить приоритеты тестовых вариантов? Допустим, вам могут предложить спроектировать электронную коммерческую систему в стиле Amazon. Естественно, нужно убедиться, что изображения продуктов появляются на правильных местах, но еще важнее, чтобы корректно осуществлялись все платежи, продукты добавлялись в очередь на отгрузку, а система не пыталась выставлять счет дважды.
- **Умение правильно совместить все части**: понимаете ли вы, как работает программное обеспечение и как оно может вписаться в большую систему? Представьте, что вам предложили протестировать электронную таблицу Google Spreadsheets. Очень важно проверить все основные операции — открытие, сохранение и редактирование документов. Однако следует помнить, что Google Spreadsheets — часть огромной системы. Вы обязательно должны проверить интеграцию с Gmail, плагинами и другими компонентами.
- **Организация**: используете вы системный подход или делаете первое, что приходит вам в голову? Некоторые кандидаты, когда их просят протестировать камеру,

просто озвучивают подряд все, что приходит им в голову. Но хороший кандидат разобьет процесс тестирования на направления, например «Фотографирование», «Управление изображением», «Настройки» и т. д. Системный подход поможет вам более тщательно сделать работу по написанию тестов.

- **Практичность**: можете ли вы разработать разумный план тестирования? Например, если пользователь сообщает, что программное обеспечение «падает» при открытии какого-либо изображения, а вы ему советуете переустановить программу — это не самый практичный совет. План тестирования должен быть выполнимым и реалистичным для компании.

Продемонстрировав владение этими навыками, вы покажете, что можете стать ценным членом группы тестирования.

Тестирование реального объекта

Некоторые кандидаты очень удивляются, когда их просят протестировать ручку. Ведь им предстоит тестировать программные продукты, не так ли? Может быть и так, но вопросы из «реального мира» все еще очень распространены. Давайте рассмотрим пример.

Вопрос: как вы будете тестировать скрепку?

Шаг 1. Кто будет ее использовать? И зачем?

Вам нужно обсудить с интервьюером, кто будет использовать продукт и с какой целью. Ответ может быть не таким, как вы ожидаете. Например: «ее будет использовать учитель, чтобы скрепить листы бумаги» или «скрепка будет использоваться дизайнером, который попытается согнуть ее в контур животного», а может, и то и другое. Ответ на вопрос сформирует ваш подход к остальным вопросам.

Шаг 2. Какие варианты использования?

Для вас будет полезно составить список возможных вариантов использования. В рассматриваемом случае скрепка используется для скрепления листов без повреждения бумаги.

Другие вопросы могут подразумевать несколько вариантов использования. Возможно, продукт должен осуществлять отправку и получение контента, запись и стирание и т. д.

Шаг 3. Каковы границы действия предмета?

Ограничением в нашем случае может быть, например, до 30 листов бумаги без деформации скрепки и до 50 листов с небольшой деформацией.

Границы распространяются и на окружающую среду. Должна ли скрепка работать при высоких температурах (30–50 градусов)? А как насчет предельно низкой температуры?

Шаг 4. Каковы условия отказа (стресс-тест)?

Ни один из продуктов не является абсолютно отказоустойчивым, поэтому анализ условий отказа — обязательная часть вашего тестирования. Обсудите

с интервьюером, когда (при каких условиях) продукт может перестать работать и что подразумевается под отказом.

Например, если вы тестируете стиральную машину, то можно предположить, что она должна оставаться работоспособной при загрузке 30 рубашек (или брюк). Но загрузка 30–45 рубашек может вызвать незначительный отказ — одежда будет плохо постирана. При загрузке более 45 рубашек возможет критичный отказ — выход машины из строя, но в этом случае под отказом следует понимать, что машина просто не запустится. Отказ безусловно *не должен* подразумевать пожар или разлив воды.

Шаг 5. Как организовать тестирование?

В некоторых случаях необходимо обсудить детали процедуры тестирования. Если вам нужно убедиться, что стул может прослужить 5 лет, не будете же вы брать его домой и ждать? Вместо этого вы должны определить, что такое «нормальное использование». (Сколько человек посидит на нем за год? Не забудьте про подлокотники...) Вполне возможно, что вы захотите автоматизировать часть работы по тестированию, а не использовать только ручной труд.

Тестирование программного обеспечения

Тестирование фрагментов программного кода подобно тестированию объектов реального мира. Главное различие заключается в том, что при тестировании программного обеспечения особое внимание уделяется подробностям самой процедуры тестирования.

Перед началом тестирования программного обеспечения нужно обратить внимание на два момента.

- **Ручное и автоматизированное тестирование**. В идеальном случае можно автоматизировать все, но мы имеем дело с реальными задачами. Некоторые вещи гораздо проще и быстрее протестировать вручную, потому что компьютер не может решить некоторые задачи качественного характера (например, может ли контент считаться порнографическим?). Не забывайте, что компьютер может распознать только то, что было запрограммировано заранее, а человек может заметить и новые проблемы, которые не были учтены при анализе. И люди, и компьютеры являются неотъемлемыми частями процесса тестирования.
- **Ящики: «черный» vs «белый»**. Термины «черный ящик»/«белый ящик» указывают на степень доступа к программному обеспечению. При тестировании методом «черного ящика» вы получаете программное обеспечение «как есть» и тестируете. «Белый ящик» предполагает, что отдельные функции тестирования можно автоматизировать. Впрочем, тестирование методом «черного ящика» также можно автоматизировать, но сделать это существенно сложнее.

Рассмотрим возможную последовательность действий от начала до конца.

Шаг 1. Какой метод использовать: «черный ящик» или «белый ящик»?

Этот вопрос можно задать и позднее, но я предпочитаю разобраться с ним в самом начале. Согласуйте с интервьюером метод тестирования — «черный ящик» и/или «белый ящик».

Шаг 2. Кто и зачем будет использовать программное обеспечение?

Обычно у программного обеспечения есть некая целевая аудитория. Например, если вы тестируете функцию родительского контроля веб-браузера, то пользователями являются родители (управление блокировкой) и дети (ограничение блокировкой). Кроме того, могут появиться «гости», которые, с одной стороны, не должны управлять блокировкой, а с другой — не должны ограничиваться ею.

Шаг 3. Каковы варианты использования?

В сценарии с блокирующим ПО варианты использования могут быть следующие: родители устанавливают программу, управляют блокировкой и сами пользуются Интернетом. Для детей варианты использования ограничены доступом к легальному контенту (хотя возможность доступа к нелегальному контенту тоже должна быть рассмотрена как отдельный вариант использования).

Помните, что варианты использования следует обязательно обсудить с интервьюером, а не придумывать самому.

Шаг 4. Каковы рамки использования?

Теперь, когда вы определили варианты использования, вы должны выяснить, что под ними подразумевается. Например, что означает блокировка сайта? Нужно блокировать только «запретную» страницу или весь сайт? Программное обеспечение должно анализировать и находить «плохой» контент или работать на основании черного и белого списков? Если программа должна самообучаться, то какой контент можно считать недопустимым, какая вероятность ложных положительных или ложных отрицательных срабатываний может считаться приемлемой?

Шаг 5. Каковы условия отказа (стресс-тест)?

Когда программное обеспечение перестанет работать, — а это непременно произойдет, — как будет выглядеть отказ? Ясно, что ошибка не должна нарушать работоспособность компьютера. Вместо этого программное обеспечение должно всего лишь разрешить доступ к заблокированному сайту или наоборот. В последнем случае можно говорить о реализации выборочного переопределения правил через пароль.

Шаг 6. Какие тестовые сценарии следует проработать? Как организовать тестирование?

В этой фазе и проявляется разница между ручным и автоматическим тестированием, между методами «черного ящика» и «белого ящика».

Шаги 3 и 4 позволяют определить приблизительные варианты использования. Шестой шаг показывает, как нужно выполнять тестирование. Какие ситуации вы будете тестировать? Какой из этих шагов можно автоматизировать, а какой требует человеческого участия?

Помните, что автоматизация позволяет провести очень серьезное тестирование, но у нее есть существенные недостатки. Ручное тестирование обязательно должно входить в процедуру тестирования, которую вы предлагаете.

Проходя по списку, не старайтесь озвучить каждый сценарий, какой вам только придет в голову. Такой подход слишком хаотичен, и вы наверняка упустите какие-нибудь ключевые категории. Действуйте методично. Разбейте программный продукт на основные компоненты и тестируйте их по отдельности. Работая в этом направлении, вы не только составите более полный список тестовых сценариев, но и покажете, что вы способны действовать системно и методично.

Тестирование функций

Тестирование функций — самый простой тип тестирования. Обычно такое тестирование сводится к проверке корректности ввода и вывода.

Это не означает, что разговор с интервьюером не столь важен. Обсудите все предположения, особенно затрагивающие работу в определенных ситуациях.

Предположим, вам предложили написать код для проверки функции `sort(int[] array)`, выполняющей сортировку массива целых чисел. Ниже представлен один из возможных планов действий.

Шаг 1. Определите тестовые сценарии

Проанализируйте следующие типы тестовых сценариев:

- *Нормальный случай.* Функция генерирует правильный результат для типичного ввода? Не забудьте о потенциальных проблемах. Например, если сортировка часто требует сегментации, разумно предположить, что алгоритм может дать сбой на массивах с нечетным количеством элементов, так как их невозможно разделить ровно пополам. В ваших тестовых сценариях должны быть учтены оба случая.
- *Крайние случаи.* Что произойдет, если массив окажется пустым? Или очень маленьким (1 элемент)? А что случится, если массив окажется слишком большим?
- *Null и недопустимый ввод.* Нужно проверить, как будет себя вести код, если пользователь введет недопустимое значение. Например, когда вы тестируете функцию, генерирующую n-число Фибоначчи, в тестовые сценарии стоит включить ситуацию с отрицательным значением n.
- *Аномальный ввод.* Также иногда встречается четвертая разновидность ввода. Что случится, если функция получит отсортированный массив? Или массив, отсортированный в обратном порядке?

Проведение таких тестов требует знания функции, которую вы пишете. Если вам неясны ограничения, обсудите их с интервьюером.

Шаг 2. Определите ожидаемый результат

Часто ожидаемый результат очевиден: правильный вывод. Однако в некоторых случаях нужно проверить множество дополнительных деталей. Например, если метод `sort` возвращает новый отсортированный массив, стоит проверить, не изменился ли исходный массив.

Шаг 3. Напишите тестовый код

Как только вы определитесь с тестовыми случаями и результатом, задача написания кода тестовых сценариев обычно становится вполне тривиальной. Код может выглядеть примерно так:

```
1  void testAddThreeSorted() {
2    MyList list = new MyList();
3    list.addThreeSorted(3, 1, 2); // Добавить 3 элемента с сортировкой
4    assertEquals(list.getElement(0), 1);
5    assertEquals(list.getElement(1), 2);
6    assertEquals(list.getElement(2), 3);
7  }
```

Поиск и устранение неисправностей

Последняя разновидность вопросов этой категории — как бы вы подошли к отладке или диагностике существующей проблемы? Много кандидатов, получив подобный вопрос, приходят в замешательство и дают нереалистичный ответ вроде «переустановлю программу». При решении этой задачи, как и многих остальных, можно применить структурный подход.

Рассмотрим конкретный пример. Вы — член команды Google Chrome — получили отчет об ошибке: при запуске браузера происходит фатальный сбой. Ваши действия?

Переустановка браузера может решить проблему этого пользователя, но вряд ли поможет другим, у которых могла возникнуть аналогичная ситуация. Ваша цель понять, что происходит *на самом деле*, чтобы разработчики могли исправить ошибку.

Шаг 1. Проанализируйте ситуацию

Прежде всего нужно задать вопросы, позволяющие максимально прояснить ситуацию:

- Как давно возникла данная проблема?
- Какая версия браузера используется? Какая операционная система?
- Проблема возникает постоянно? Как часто она возникает? Когда именно?
- *Выводится* ли сообщение об ошибке?

Шаг 2. Разбейте задачу на части

Теперь, когда вам ясны детали сценария, можно попробовать разбить задачу на части, пригодные для модульного тестирования. В нашем случае:

1. Откройте стартовое меню Windows.
2. Щелкните на значке Chrome.
3. Запустите браузер.
4. Браузер загружает настройки.
5. Браузер отправляет HTTP-запрос на получение домашней страницы.
6. Браузер получает HTTP-ответ.
7. Браузер разбирает веб-страницу.

8. Браузер выводит контент на экран.

В этом процессе что-то перестало работать, что и вызвало отказ браузера. Хороший тестер пройдется по всем этапам сценария и выяснит причину.

Шаг 3. Создайте конкретные управляемые тесты

К каждому из перечисленных компонентов должны быть приложены практические инструкции — то, что вы попросите сделать пользователей или сможете сделать сами (скажем, повторить их действия на своей машине). В реальном мире вы будете иметь дело с клиентами; не давайте им инструкции, которые они не смогут (или не захотят) выполнить.

Вопросы собеседования

11.1. Найдите ошибку (ошибки) в следующем коде:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3 printf("%d\n", i);
```

Подсказки: 257, 299, 362

11.2. Вам предоставили исходный код приложения, которое аварийно завершается после запуска. После десяти запусков в отладчике вы обнаруживаете, что каждый раз программа завершается в разных местах. Приложение однопотоковое и использует только стандартную библиотеку C. Какие ошибки могут вызвать сбой приложения? Как вы организуете тестирование?

Подсказки: 325

11.3. В компьютерной реализации шахмат имеется метод `boolean canMoveTo(int x, int y)`. Этот метод (часть класса `Piece`) возвращает результат, по которому можно понять, возможно ли перемещение фигуры на позицию `(x, y)`. Объясните, как вы будете тестировать данный метод.

Подсказки: 329, 401

11.4. Как вы проведете нагрузочный тест веб-страницы без использования специальных инструментов тестирования?

Подсказки: 313, 345

11.5. Как вы организуете тестирование авторучки?

Подсказки: 140, 164, 220

11.6. Как вы организуете тестирование банкомата, подключенного к распределенной банковской системе?

Подсказки: 210, 225, 268, 349, 393

Подсказки начинаются на с. 699 (скачайте ч. XIII на сайте изд-ва «Питер»).

12

C и C++

Хороший интервьюер не станет требовать, чтобы вы написали код на языке, которого не знаете. Если вам предложено написать код на C++, значит, вы упомянули его в резюме. Если вы не помните все API, не волнуйтесь — для большинства интервьюеров (но не для всех) это не важно. Однако вам стоит изучить базовый синтаксис C++ так, чтобы вы могли легко отвечать на вопросы по C/C++.

Классы и наследование

Классы C++ по своим характеристикам почти не отличаются от классов в других языках программирования, но мы все же рассмотрим их синтаксис.

Следующий код демонстрирует реализацию базового класса с наследованием.

```
#include <iostream>
using namespace std;

#define NAME_SIZE 50 // Определение макроса

class Person {
  int id; // По умолчанию все члены класса являются приватными
  char name[NAME_SIZE];

  public:
   void aboutMe() {
     cout << "I am a person.";
   }
};

class Student : public Person {
 public:
  void aboutMe() {
    cout << "I am a student.";
  }
};

int main() {
  Student * p = new Student();
  p->aboutMe(); // Выводит текст "I am a student."
  delete p; // Важно! Не забудьте освободить выделенную память.
  return 0;
}
```

По умолчанию в C++ все поля класса и методы являются приватными. Для изменения их уровня видимости используется ключевое слово `public`.

Конструкторы и деструкторы

Конструктор класса автоматически вызывается при создании объекта. Если конструктор не определен, компилятор автоматически генерирует так называемый конструктор по умолчанию. Также вы можете определить собственный конструктор.

Если вызов конструктора ограничивается инициализацией примитивных типов, проще всего это сделать так:

```
1 Person(int a) {
2     id = a;
3 }
```

Такой способ работает для примитивных типов, но также возможно использовать другой синтаксис:

```
1 Person(int a) : id(a) {
2     ...
3 }
```

Поле класса `id` инициализируется перед созданием объекта и перед выполнением оставшейся части конструктора. В частности, такая возможность пригодится для создания полей, содержащих константы, значения которых присваиваются только один раз.

Деструктор удаляет объект и автоматически вызывается при уничтожении объекта. Деструктору нельзя передать аргумент, так как он не вызывается явно:

```
1  ~Person() {
2    delete obj; // Освобождение памяти, выделенной для name
3  }
```

Виртуальные функции

В приведенном выше примере определялся новый объект `p` класса `Student`:

```
1 Student * p = new Student();
2 p->aboutMe();
```

А что случится, если определить `p` с типом `Person *`?

```
1 Person * p = new Student();
2 p->aboutMe();
```

В этом случае будет выведен текст `"I am a person"`. Это связано с тем, что вызываемая функция `aboutMe` выбирается во время компиляции. Такой механизм называется *статической компоновкой*.

Если вы хотите, чтобы была вызвана реализация `aboutMe` из класса `Student`, функцию `aboutMe` в классе `Person` следует объявить виртуальной (`virtual`):

```
1  class Person {
2    ...
3    virtual void aboutMe() {
4      cout << "I am a person.";
5    }
6  };
7
```

```
8  class Student : public Person {
9   public:
10    void aboutMe() {
11      cout << "I am a student.";
12    }
13  };
```

У виртуальных функций есть и другое применение: если вы не можете (или не хотите) реализовать метод в родительском классе. Представьте, например, что классы `Student` и `Teacher` должны быть производными от класса `Person`, чтобы в них можно было определить общий метод `addCourse(string s)`. Однако вызов `addCourse` для класса `Person` не имеет смысла, поскольку реализация этого метода зависит от фактического типа объекта — `Student` или `Teacher`.

В этом случае `addCourse` определяется в классе `Person` как виртуальная функция, реализация которой доверяется субклассу:

```
1  class Person {
2    int id; // По умолчанию все члены класса являются приватными
3    char name[NAME_SIZE];
4    public:
5    virtual void aboutMe() {
6      cout << "I am a person." << endl;
7    }
8    virtual bool addCourse(string s) = 0;
9  };
10
11  class Student : public Person {
12    public:
13    void aboutMe() {
14      cout << "I am a student." << endl;
15    }
16
17    bool addCourse(string s) {
18      cout << "Added course " << s << " to student." << endl;
19      return true;
20    }
21  };
22
23  int main() {
24    Person * p = new Student();
25    p->aboutMe(); // Выводит текст "I am a student."
26    p->addCourse("History");
27    delete p;
28  }
```

Заметьте, что с определением `addCourse` как чисто виртуальной функции класс `Person` становится абстрактным классом; экземпляры таких классов создаваться не могут.

Виртуальный деструктор

Виртуальная функция становится естественным шагом к концепции «виртуального деструктора». Допустим, вы хотите реализовать деструкторы для `Person` и `Student`. Наивное решение может выглядеть так:

```
class Person {
 public:
  ~Person() {
    cout << "Deleting a person." << endl;
  }
};

class Student : public Person {
 public:
  ~Student() {
    cout << "Deleting a student." << endl;
  }
};

int main() {
  Person * p = new Student();
  delete p; // выводит текст "Deleting a person."
}
```

Как и в предыдущем примере, поскольку p является объектом класса `Person`, будет вызван деструктор этого класса. Но это создаст проблемы, поскольку память `Student` может остаться не освобожденной.

Проблема решается простым объявлением деструктора `Person` виртуальным:

```
class Person {
 public:
  virtual ~Person() {
    cout << "Deleting a person." << endl;
  }
};

class Student : public Person {
 public:
  ~Student() {
    cout << "Deleting a student." << endl;
  }
};

int main() {
  Person * p = new Student();
  delete p;
}
```

Выводится следующий результат:

```
Deleting a student.
Deleting a person.
```

Значения по умолчанию

Аргументам функций можно задавать значения по умолчанию. Обратите внимание: параметры со значениями по умолчанию должны располагаться в правой части объявления функции, иначе при вызове будет неясно, какие параметры отсутствуют в списке:

```
int func(int a, int b = 3) {
  x = a;
```

```
3    y = b;
4    return a + b;
5  }
6
7  w = func(4);
8  z = func(4, 5);
```

Перегрузка операторов

Механизм перегрузки операторов позволяет применять любые операторы (например, +) к объектам, которые изначально эту операцию не поддерживали. Например, если вам потребуется объединить два объекта `BookShelves` в один, вы можете перегрузить оператор +:

```
1  BookShelf BookShelf::operator+(BookShelf &other) { ... }
```

Указатели и ссылки

Указатель хранит адрес переменной и может использоваться для осуществления любых операций, которые могут использоваться с переменной напрямую (например, чтение и изменение ее значения).

Два указателя могут быть идентичными, так что изменение значения по одному из них приведет к изменению значения по другому (поскольку они ссылаются на один и тот же адрес памяти):

```
1  int * p = new int;
2  *p = 7;
3  int * q = p;
4  *p = 8;
5  cout << *q; // Выводит 8
```

Размер указателя зависит от архитектуры: 32 бита (на 32-битной машине) или 64 бита (на 64-битной). Это важный момент, поскольку интервьюеры обычно интересуются, сколько памяти занимает структура данных.

Ссылки

Ссылка — это альтернативное имя (псевдоним) существующего объекта. Со ссылкой не связывается собственный блок памяти. Например:

```
1  int a = 5;
2  int & b = a;
3  b = 7;
4  cout << a; //Выводит 7
```

Во второй строке `b` — это ссылка на `a`. Изменение `b` приводит к изменению `a`.

Ссылку нельзя создать без указания конкретного блока памяти, на который она будет ссылаться. Однако вы можете создать ссылку, указывающую на конкретное значение (без переменной):

```
1 /* выделяем место в памяти для значения 12 и назначаем
2 b ссылкой на этот блок памяти */
3 const int & b = 12;
```

В отличие от указателей, ссылки не могут принимать значение NULL и не могут переназначаться на другой блок памяти.

Вычисления с указателями

В программах часто используется суммирование указателей, например:

```
1 int * p = new int[2];
2 p[0] = 0;
3 p[1] = 1;
4 p++;
5 cout << *p; // выведет 1
```

Операция p++ пропускает sizeof(int) байт, поэтому код выводит 1. Если бы элементы p относились к другому типу, переход осуществлялся на столько байтов, сколько занимает структура данных.

Шаблоны

Шаблоны (templates) — механизм повторного использования кода для одного класса с разными типами данных. Представьте, что существует структура данных — аналог списка, которую нужно использовать для списков разных типов. Следующий код реализует шаблонный класс ShiftedList:

```
1  template <class T>class ShiftedList {
2    T* array;
3    int offset, size;
4   public:
5    ShiftedList(int sz) : offset(0), size(sz) {
6      array = new T[size];
7    }
8
9    ~ShiftedList() {
10     delete [] array;
11   }
12
13   void shiftBy(int n) {
14     offset = (offset + n) % size;
15   }
16
17   T getAt(int i) {
18     return array[convertIndex(i)];
19   }
20
21   void setAt(T item, int i) {
22     array[convertIndex(i)] = item;
23   }
24
25 private:
26   int convertIndex(int i) {
27     int index = (i - offset) % size;
28     while (index < 0) index += size;
29     return index;
30   }
31 };
```

Вопросы собеседования

12.1. Напишите на C++ метод для вывода последних k строк входного файла.

Подсказки: 449, 459

12.2. Реализуйте на C или C++ функцию `void reverse(char* str)` для перестановки символов строки, завершенной нуль-символом, в обратном порядке.

Подсказки: 410, 452

12.3. Проведите сравнительный анализ хеш-таблицы и ассоциативного массива из стандартной библиотеки шаблонов (STL). Как реализована хеш-таблица? Если объем данных невелик, какие структуры данных можно использовать вместо хеш-таблицы?

Подсказки: 423

12.4. Как работают виртуальные функции в C++?

Подсказки: 463

12.5. Чем глубокое копирование отличается от поверхностного? Объясните, как использовать эти виды копирования.

Подсказки: 445

12.6. Каково назначение ключевого слова `volatile` в C?

Подсказки: 456

12.7. Почему деструктор базового класса должен объявляться виртуальным?

Подсказки: 421, 460

12.8. Напишите метод, получающий указатель на структуру `Node` в параметре и возвращающий полную копию переданной структуры данных. Структура данных `Node` содержит два указателя на другие узлы (другие структуры `Nodes`).

Подсказки: 427, 462

12.9. Напишите класс «умного указателя» — типа данных (обычно реализованного на базе шаблона), моделирующего указатель с поддержкой автоматической уборки мусора. Умный указатель автоматически подсчитывает количество ссылок на объект `SmartPointer<T*>` и освобождает объект типа `T`, когда число ссылок становится равным 0.

Подсказки: 402, 438, 453

12.10. Напишите функции динамического выделения и освобождения памяти, которые работают с памятью таким образом, что возвращаемый адрес памяти кратен степени 2.

Пример:

`align_malloc(1000,128)` возвращает адрес памяти, который является кратным 128 и указывает на блок памяти размером 1000 байт.

Функция `aligned_free()` освобождает память, выделенную с помощью `align_malloc`.

Подсказки: 413, 432, 440

12.11. Напишите на C функцию (`my2DAlloc`), которая выделяет память для двумерного массива. Минимизируйте количество вызовов `malloc` и примите меры к тому, чтобы для обращения к памяти можно было использовать синтаксис `arr[i] j]`.

Подсказки: 406, 418, 426

Дополнительные вопросы: связные списки (2.6), тестирование (11.1), Java (13.4), потоки и блокировки (15.3).

Подсказки начинаются на с. 715 (скачайте ч. XIII на сайте изд-ва «Питер»).

13

Java

Вопросы, относящиеся к Java, не раз встречаются в книге, но эта глава полностью посвящена вопросам по языку Java и его синтаксису. Такие вопросы реже встречаются на собеседованиях в крупных компаниях, которые любят проверять способности, а не знания кандидата (таким корпорациям хватает времени и ресурсов, чтобы обучить кандидатов конкретному языку программирования). Однако в некоторых компаниях любят задавать подобные вопросы.

Подход к изучению

Вопросами по синтаксису проверяются знания, и поэтому на первый взгляд специального подхода к подготовке быть не может. В конце концов, нужно просто знать правильный ответ, не так ли?

И да, и нет. Конечно, лучше всего просто изучить Java вдоль и поперек. Но если вы все же попадете впросак, попробуйте действовать так:

1. Создайте примерный сценарий и спросите себя, что должно происходить на уровне здравого смысла.
2. Подумайте, как такой сценарий обрабатывается в других языках.
3. Подумайте, как бы вы подошли к разрешению подобной ситуации на месте разработчика языка. К каким последствиям приведет каждое принятое решение?

Ответ, к которому вы придете методом логических рассуждений, может произвести не меньшее (а то и большее) впечатление на интервьюера, чем ответ, который вам известен заранее. В любом случае, не пытайтесь блефовать. Скажите: «Я не уверен, что могу вспомнить точный ответ, но хочу попробовать его вывести. Предположим, что у нас есть этот код...»

Перегрузка vs переопределение

Перегрузка (overloading) — термин, используемый для описания ситуации, когда два метода с одним именем отличаются типами или количеством аргументов:

```
1 public double computeArea(Circle c) { ... }
2 public double computeArea(Square s) { ... }
```

Переопределение (overriding) происходит тогда, когда метод имеет такое же имя и сигнатуру, как и другой метод в суперклассе.

```
1  public abstract class Shape {
2    public void printMe() {
3      System.out.println("I am a shape.");
```

```
4    }
5    public abstract double computeArea();
6  }
7
8  public class Circle extends Shape {
9    private double rad = 5;
10   public void printMe() {
11     System.out.println("I am a circle.");
12   }
13
14   public double computeArea() {
15     return rad * rad * 3.15;
16   }
17 }
18
19 public class Ambiguous extends Shape {
20   private double area = 10;
21   public double computeArea() {
22     return area;
23   }
24 }
25
26 public class IntroductionOverriding {
27   public static void main(String[] args) {
28     Shape[] shapes = new Shape[2];
29     Circle circle = new Circle();
30     Ambiguous ambiguous = new Ambiguous();
31
32     shapes[0] = circle;
33     shapes[1] = ambiguous;
34
35     for (Shape s : shapes) {
36       s.printMe();
37       System.out.println(s.computeArea());
38     }
39   }
40 }
```

Вывод:

```
1  I am a circle.
2  78.75
3  I am a shape.
4  10.0
```

Заметьте, что класс `Circle` переопределил метод `printMe()`, тогда как в `Ambiguous` этот метод остался без переопределения.

Java Collection Framework

Java Collection Framework — очень полезный набор взаимосвязанных классов и интерфейсов. Эти классы и интерфейсы еще встретятся вам в книге, а пока рассмотрим его наиболее полезные элементы.

`ArrayList` — массив с динамически изменяемым размером, автоматически расширяющийся при вставке новых элементов.

```
1 ArrayList<String> myArr = new ArrayList<String>();
2 myArr.add("one");
3 myArr.add("two");
4 System.out.println(myArr.get(0)); /* выводит <one> */
```

`Vector` очень похож на `ArrayList`, но является синхронизированным. Синтаксис работы также практически не отличается.

```
1 Vector<String> myVect = new Vector<String>();
2 myVect.add("one");
3 myVect.add("two");
4 System.out.println(myVect.get(0));
```

`LinkedList` используется для работы со связными списками в коде Java. Об этом классе редко вспоминают на собеседованиях, но о нем полезно знать, потому что в операциях с этим классом демонстрируется синтаксис работы с итераторами.

```
1 LinkedList<String> myLinkedList = new LinkedList<String>();
2 myLinkedList.add("two");
3 myLinkedList.addFirst("one");
4 Iterator<String> iter = myLinkedList.iterator();
5 while (iter.hasNext()) {
6     System.out.println(iter.next());
7 }
```

`HashMap` широко используется как на собеседовании, так и в реальной работе. Пример синтаксиса:

```
1 HashMap<String, String> map = new HashMap<String, String>();
2 map.put("one", "uno");
3 map.put("two", "dos");
4 System.out.println(map.get("one"));
```

Перед собеседованием убедитесь, что вы очень хорошо понимаете этот синтаксис. Он вам пригодится.

Вопросы собеседования

Практически все решения в этой книге написаны на Java, поэтому вопросов для этой главы немного. Кроме того, большинство этих вопросов проверяет ваше знание тонкостей синтаксиса, так как вопросов по программированию на Java в книге и так хватает.

13.1. Как повлияет на наследование объявление конструктора приватным?

Подсказки: 404

13.2. Будет ли выполняться блок `finally` (в коде Java), если оператор `return` находится внутри `try`-блока (конструкция `try-catch-finally`)?

Подсказки: 409

13.3. В чем разница между `final`, `finally` и `finalize`?

Подсказки: 412

13.4. Объясните разницу между шаблонами C++ и обобщениями (generics) в языке Java.

Подсказки: 416, 425

13.5. Объясните различия между `TreeMap`, `HashMap` и `LinkedHashMap`. Приведите пример ситуации, в которой каждая из этих коллекций работает лучше других.

Подсказки: 420, 424, 430, 454

13.6. Объясните, что такое рефлексия (reflection) объектов в Java и для чего она используется.

Подсказки: 435

13.7. Существует класс `Country`, содержащий методы `getContinent()` и `getPopulation()`. Напишите функцию `int getPopulation(List<Country> countries, String continent)`, которая вычисляет население заданного континента по списку всех стран и названию континента с использованием лямбда-выражений.

Подсказки: 448, 461, 464

13.8. Напишите функцию `List<Integer> getRandomSubset(List<Integer> list)`, возвращающую случайное подмножество произвольного размера. Все подмножества (включая пустое) должны выбираться с одинаковой вероятностью. При написании функции следует использовать лямбда-выражения.

Подсказки: 443, 450, 457

Дополнительные вопросы: массивы и строки (1.3), объектно-ориентированное проектирование (7.12), потоки и блокировки (15.3).

Подсказки начинаются на с. 713 (скачайте ч. XIII на сайте изд-ва «Питер»).

14

Базы данных

Если в вашем резюме будет заявлен опыт работы с базами данных, ждите вопросов по этой теме. Мы рассмотрим некоторые ключевые понятия и проведем краткий обзор решения задач такого типа. Не удивляйтесь незначительным различиям в синтаксисе приведенных запросов. Существует множество диалектов SQL; возможно, вы работали с другой версией языка. Все примеры, приведенные в этой книге, были протестированы в Microsoft SQL Server.

Синтаксис SQL и его варианты

Ниже приведены примеры явного (explicit) и неявного (implicit) соединения. Эти две команды эквивалентны, а выбор определяется личными предпочтениями. Для единства стиля мы будем использовать явное соединение.

```
1 /* Явное соединение */
2 SELECT CourseName, TeacherName
3     FROM Courses INNER JOIN Teachers
4     ON Courses.TeacherID = Teachers.TeacherID
5
6 /* Неявное соединение */
7 SELECT CourseName, TeacherName
8     FROM Courses, Teachers
9     WHERE Courses.TeacherID = Teachers.TeacherID
```

Денормализованные и нормализованные базы данных

Нормализованные базы данных минимизируют избыточность, а денормализованные базы данных проектируются для оптимизации времени чтения.

В традиционной нормализованной базе данных с данными вида `Courses` (Курсы) и `Teachers` (Преподаватели) используется `TeacherID` — ключ таблицы `Teachers`. Преимущество данного решения в том, что вся информация о преподавателе (имя, адрес и т. д.) хранится в базе данных только в одном экземпляре (для одного преподавателя), а недостаток — дополнительные затраты ресурсов на соединение.

В денормализованных базах данных хранятся избыточные записи. Например, если приведенный выше запрос часто повторяется, то целесообразно будет сохранить имя преподавателя в таблице `Courses`. Денормализованные базы данных используются для создания хорошо масштабируемых систем.

Команды SQL

Рассмотрим основной синтаксис SQL на примере упомянутой ранее базы данных. База данных имеет очень простую структуру (* обозначает первичный ключ):

```
Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName
StudentCourses: CourseID*, StudentID*
```

Реализуем некоторые запросы, используя вышеприведенные таблицы.

Запрос 1. Регистрация студента

Первый запрос должен выводить список всех студентов и количество курсов, которые посещает каждый студент.

На первый взгляд команда могла бы выглядеть примерно так:

```
1  /* Неправильный код */
2  SELECT Students.StudentName, count(*)
3  FROM Students INNER JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID
```

У этого решения три недостатка:

1. Мы исключили студентов, которые не зарегистрировались ни на каких курсах, поскольку `StudentCourses` содержит только зарегистрированных студентов. Нужно изменить соединение на `LEFT JOIN`.
2. Даже если изменить соединение на `LEFT JOIN`, запрос все равно не станет правильным. Вызов `count(*)` вернет количество элементов в заданной группе идентификаторов (`StudentID`). Но даже если студент посещает 0 курсов, он получит единичку в группе. Нам нужно изменить вызов `count` на `count(StudentCourses.CourseID)`.
3. Мы выполнили группировку по `Students.StudentID`, но в каждой группе остаются повторяющиеся `StudentName`. Как база данных узнает, какое значение `StudentName` вернуть? Конечно, все эти значения могут быть одинаковыми, но база данных не понимает этого. Мы должны применить функцию-*агрегатор*, например `first(Students.StudentName)`.

Таким образом, исправленный запрос примет вид:

```
1  /* Решение 1: вспомогательный запрос */
2  SELECT StudentName, Students.StudentID, Cnt
3  FROM (
4    SELECT  Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
5    FROM Students LEFT JOIN StudentCourses
6    ON Students.StudentID = StudentCourses.StudentID
7    GROUP BY Students.StudentID
8  ) T INNER JOIN Students on T.studentID = Students.StudentID
```

Посмотрев на этот код, вы можете спросить, почему бы просто не выбрать имя студента в строке 3, чтобы строки 3–6 не приходилось включать в дополнительный запрос. Это (ошибочное) решение приведено ниже.

```
1  /* Неправильный код */
1  SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
2  FROM Students LEFT JOIN StudentCourses
3  ON Students.StudentID = StudentCourses.StudentID
4  GROUP BY Students.StudentID
```

Так поступать *нельзя*, по крайней мере не в том виде, как показано. Можно выбирать только значения, находящиеся в функции-агрегаторе или же в условии GROUP BY.

Эту же задачу можно решить иначе:

```
1  /* Решение 2: Добавление StudentName в условие GROUP BY. */
2  SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
3  FROM Students LEFT JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID, Students.StudentName
```

или

```
1  /* Решение 3: функция-агрегатор. */
2  SELECT  max(StudentName) as [StudentName], Students.StudentID,
3        count(StudentCourses.CourseID) as [Count]
4  FROM Students LEFT JOIN StudentCourses
5  ON Students.StudentID = StudentCourses.StudentID
6  GROUP BY Students.StudentID
```

Запрос 2. Размер аудитории

Реализуйте запрос для получения списка всех преподавателей и количества студентов, которым преподает каждый из них. Если один студент учится у одного преподавателя на двух курсах, он должен быть учтен дважды. Отсортируйте список в порядке убывания количества студентов.

Этот запрос можно построить шаг за шагом. Прежде всего, нужно получить список TeacherID и узнать, сколько студентов связано с каждым TheacherID. Это очень напоминает предыдущий пример.

```
1  SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
2  FROM Courses INNER JOIN StudentCourses
3  ON Courses.CourseID = StudentCourses.CourseID
4  GROUP BY Courses.TeacherID
```

Обратите внимание: INNER JOIN не выбирает преподавателей, не ведущих учебные курсы. В следующем запросе мы это исправим, так как хотим получить список всех преподавателей.

```
1  SELECT TeacherName, isnull(StudentSize.Number, 0)
2  FROM Teachers LEFT JOIN
3       (SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
4        FROM Courses INNER JOIN StudentCourses
5        ON Courses.CourseID = StudentCourses.CourseID
6        GROUP BY Courses.TeacherID) StudentSize
7  ON Teachers.TeacherID = StudentSize.TeacherID
8  ORDER BY StudentSize.Number DESC
```

Обратите внимание на обработку значений NULL в SELECT: NULL преобразуются в нули.

Проектирование небольшой базы данных

На собеседовании вас могут попросить спроектировать собственную базу данных. Сейчас мы разберем этот процесс шаг за шагом. Обратите внимание на сходство этого метода с методами объектно-ориентированного проектирования.

Шаг 1. Уберите неоднозначности

В вопросах по базам данных часто присутствуют неоднозначности (намеренные или случайные). Прежде чем начать проектировать базу данных, вам нужно разобраться с этими проблемами.

Допустим, вас попросили разработать систему хранения информации для агентства по аренде недвижимости. Вы должны узнать, сколько офисов у этого агентства. Обсудите с интервьюером степень обобщенности. Один человек вряд ли будет арендовать две квартиры в одном и том же доме (хотя и такое может произойти), но это не означает, что система должна выдавать ошибку на подобной задаче. Некоторые крайне редкие условия лучше всего реализуются обходными решениями (например, дублированием контактной информации человека в БД).

Шаг 2. Определите ключевые объекты

Затем определите базовые объекты системы, которые (чаще всего) следует преобразовать в таблицу. В нашем случае базовыми объектами будут `Property` (Свойство), `Building` (Здание), `Apartment` (Апартаменты), `Tenant` (Арендатор) и `Manager` (Менеджер).

Шаг 3. Проанализируйте отношения

Выделение базовых объектов позволяет понять, какие таблицы должны использоваться в системе. Как эти таблицы связаны друг с другом? Какие отношения при этом используются: «многие-ко-многим» или «один-ко-многим»?

У зданий с квартирами будет использоваться связь «один-ко-многим» (одно здание — много квартир), которую можно представить следующим образом:

Apartments	
ApartmentID	int
ApartmentAddress	varchar(100)
BuildingID	int

Buildings	
BuildingID	int
BuildingName	varchar(100)
BuildingAddress	varchar(500)

Обратите внимание: таблица квартир (`Apartments`) связана с таблицами зданий через поле `BuildingID`.

Если мы хотим учесть ситуацию, когда один и тот же человек арендует больше чем одну квартиру, используйте отношение «многие-ко-многим» следующим образом:

TenantApartments	
TenantID	int
ApartmentID	int

Apartments	
ApartmentID	int
ApartmentAddress	varchar(500)
BuildingID	int

Tenants	
TenantID	int
TenantName	varchar(100)
TenantAddress	varchar(500)

В таблице `TenantApartments` хранятся отношения между `Tenants` и `Apartments`.

Шаг 4. Исследуйте действия

Осталось только разобраться с подробностями. Перебрите основные операции, которые планируется осуществлять, и продумайте сохранение и чтение данных. Не забудьте, что вам придется обрабатывать информацию об условиях аренды, ценах, выездах на осмотр и т. д. Каждое из этих действий потребует дополнительных таблиц и столбцов.

Проектирование больших баз данных

Когда проектируется большая масштабируемая база данных, использованные в предыдущих примерах соединения (join) работают очень медленно. Данные придется денормализовать. Подумайте о том, как будут использоваться данные — возможно, их придется продублировать в нескольких таблицах.

Вопросы собеседования

Вопросы 1–3 относятся к следующей схеме базы данных.

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

У квартиры (Apartments) может быть несколько арендаторов (Tenants), а арендатор может арендовать несколько квартир. Квартира может находиться только в одном здании (Buildings), а здание — в одном комплексе (Complexes).

14.1. Напишите SQL-запрос для получения списка арендаторов, которые снимают более одной квартиры.

Подсказки: 408

14.2. Напишите SQL-запрос для получения списка всех зданий и количества открытых запросов (запросов со статусом `'open'`).

Подсказки: 411

14.3. Дом № 11 находится на капитальном ремонте. Напишите запрос, который закрывает все запросы на квартиры в этом здании.

Подсказки: 431

14.4. Какие существуют типы соединений? Объясните, чем они различаются и почему определенные типы лучше подходят для конкретных ситуаций.

Подсказки: 451

14.5. Что такое денормализация? Поясните ее достоинства и недостатки.

Подсказки: 444, 455

14.6. Нарисуйте диаграмму отношений для базы данных, в которой фигурируют компании, клиенты и сотрудники компаний.

Подсказки: 436

14.7. Разработайте простую базу данных, содержащую данные об успеваемости студентов, и напишите запрос, возвращающий список лучших студентов (лучшие 10 %), отсортированный по их среднему баллу.

Подсказки: 428, 442

Дополнительные вопросы: объектно-ориентированное проектирование (7.7), масштабируемость и проектирование систем (9.6).

Подсказки начинаются на с. 713 (скачайте ч. XIII на сайте изд-ва «Питер»).

15

Потоки и блокировки

Маловероятно, чтобы на собеседовании в Microsoft, Google или Amazon вам предложили реализовать многопоточный алгоритм (если только вы не будете работать в группе, для которой этот вопрос действительно актуален). Однако интервьюеры всегда стараются оценить, насколько вы разбираетесь в работе потоков, а особенно во взаимных блокировках (deadlocks).

Эта глава познакомит вас с данной темой.

Потоки в Java

Каждый поток в Java создается и управляется уникальным объектом класса `java.lang.Thread`. При запуске приложения для выполнения метода `main()` автоматически создается пользовательский поток, который называется главным потоком.

В Java потоки реализуются двумя способами:

- через реализацию интерфейса `java.lang.Runnable`;
- через расширение класса `java.lang.Thread`.

Далее будут рассмотрены оба способа.

Реализация интерфейса Runnable

Интерфейс `Runnable` имеет очень простую структуру:

```
1 public interface Runnable {
2     void run();
3 }
```

Для создания и использования потока с этим интерфейсом нужно:

1. Создать класс, который реализует интерфейс `Runnable`.
2. Создать объект типа `Thread`, передав `Runnable`-объект в аргументе конструктора `Thread`. Объект `Thread` теперь располагает `Runnable`-объектом, который реализует метод `run()`.
3. Метод `start()` вызывается для объекта `Thread`, созданного на предыдущем шаге.

Пример:

```
1  public class RunnableThreadExample implements Runnable {
2    public int count = 0;
3
4    public void run() {
5      System.out.println("RunnableThread starting.");
6      try {
```

```
7        while (count < 5) {
8          Thread.sleep(500);
9          count++;
10       }
11       } catch (InterruptedException exc) {
12         System.out.println("RunnableThread interrupted.");
13     }
14       System.out.println("RunnableThread terminating.");
15    }
16  }
17
18  public static void main(String[] args) {
19    RunnableThreadExample instance = new RunnableThreadExample();
20    Thread thread = new Thread(instance);
21    thread.start();
22
23    /* ожидает, пока поток досчитает до 5 (с замедлением) */
24    while (instance.count != 5) {
25      try {
26        Thread.sleep(250);
27      } catch (InterruptedException exc) {
28        exc.printStackTrace();
29     }
30    }
31  }
```

Обратите внимание: фактически в этом коде нужно только реализовать наш метод `run()` (см. строку 4). Тогда другой метод сможет передать экземпляр класса новому потоку `new Thread(obj)` (строки 19–20) и вызвать `start()` потока (см. строку 21).

Расширение класса Thread

Другой способ — расширение класса `Thread`. Это почти всегда означает, что мы переопределяем метод `run()`. Тогда субкласс сможет явно вызвать конструктор потока в своем конструкторе.

Пример расширения класса `Thread`:

```
1  public class ThreadExample extends Thread {
2    int count = 0;
3
4    public void run() {
5      System.out.println("Thread starting.");
6      try {
7        while (count < 5) {
8          Thread.sleep(500);
9          System.out.println("In Thread, count is " + count);
10         count++;
11     }
12     } catch (InterruptedException exc) {
13       System.out.println("Thread interrupted.");
14    }
15     System.out.println("Thread terminating.");
16   }
17 }
18
19 public class ExampleB {
```

```
20    public static void main(String args[]) {
21      ThreadExample instance = new ThreadExample();
22      instance.start();
23
24      while (instance.count != 5) {
25      try {
26          Thread.sleep(250);
27        } catch (InterruptedException exc) {
28          exc.printStackTrace();
29      }
30     }
31    }
32  }
```

Этот код в целом похож на предыдущий пример; отличается он только тем, что мы расширяем класс `Thread` вместо того, чтобы реализовывать интерфейс, что позволяет вызвать `start()` непосредственно для экземпляра класса.

Расширение класса Thread и реализация Runnable-интерфейса

При создании потоков существуют две причины, по которым реализация `Runnable`-интерфейса предпочтительнее расширения класса `Thread`:

1. Java не поддерживает множественное наследование. Следовательно, класс, расширяющий `Thread`, не сможет расширять любой другой класс. Класс, расширяющий интерфейс `Runnable`, может расширять другой класс.
2. Возможно, программиста интересует только возможность запуска на выполнение, и полный набор возможностей, связанных с наследованием от `Thread`, окажется избыточным.

Синхронизация и блокировки

Потоки в пределах заданного процесса используют пространство памяти совместно. У такого подхода есть как преимущества, так и недостатки: потоки могут совместно использовать данные, что может быть весьма полезно. С другой стороны, появляется опасность конфликтов, когда ресурс одновременно изменяется двумя потоками. Для управления доступом к общим ресурсам в Java используется механизм *синхронизации*.

Ключевые слова `synchronized` и `lock` — основа для реализации синхронного выполнения кода.

Синхронизация методов

Обычно для ограничения доступа к совместно используемым ресурсам используется ключевое слово `synchronized`. Оно может применяться к методам и блокам кода, предотвращая одновременный доступ множества потоков к *одному и тому же* объекту.

Для наглядности рассмотрим следующий код:

```
1  public class MyClass extends Thread  {
2    private String name;
3    private MyObject myObj;
```

```
4
5    public MyClass(MyObject obj, String n) {
6      name = n;
7      myObj = obj;
8    }
9
10   public void run() {
11     myObj.foo(name);
12   }
13 }
14
15 public class MyObject {
16   public synchronized void foo(String name) {
17     try {
18       System.out.println("Thread " + name + ".foo(): starting");
19       Thread.sleep(3000);
20       System.out.println("Thread " + name + ".foo(): ending");
21     } catch (InterruptedException exc) {
22       System.out.println("Thread " + name + ": interrupted.");
23    }
24   }
25 }
```

Могут ли два экземпляра `MyClass` вызвать `foo` одновременно? Если у них есть тот же самый экземпляр `MyObject` — нет. Но если они содержат различные ссылки, то ответ будет положительным.

```
1  /* Разные ссылки - оба потока могут вызвать MyObject.foo() */
2  MyObject obj1 = new MyObject();
3  MyObject obj2 = new MyObject();
4  MyClass thread1 = new MyClass(obj1, "1");
5  MyClass thread2 = new MyClass(obj2, "2");
6  thread1.start();
7  thread2.start()
8
9  /* Одна ссылка на obj. Только один поток сможет вызвать foo,
10   * а другому придется ожидать. */
11 MyObject obj = new MyObject();
12 MyClass thread1 = new MyClass(obj, "1");
13 MyClass thread2 = new MyClass(obj, "2");
14 thread1.start()
15 thread2.start()
```

Статические методы синхронизируются по *блокировке класса*. Два потока из предыдущего кода не смогли бы одновременно выполнить синхронизированные статические методы одного класса, даже если один вызывал `foo`, а второй — `bar`.

```
1  public class MyClass extends Thread  {
2    ...
3    public void run() {
4      if (name.equals("1")) MyObject.foo(name);
5      else if (name.equals("2")) MyObject.bar(name);
6    }
7  }
8
```

```
9  public class MyObject {
10    public static synchronized void foo(String name) {/* Как и прежде */}
11    public static synchronized void bar(String name) { /* Как для foo */ }
12  }
```

При запуске этого кода будет выведен следующий результат:

```
Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

Синхронизованные блоки кода

Также возможно синхронизировать блоки кода. Эта операция имеет очень много общего с синхронизацией методов.

```
1  public class MyClass extends Thread  {
2    ...
3    public void run() {
4      myObj.foo(name);
5    }
6  }
7  public class MyObject {
8    public void foo(String name) {
9      synchronized(this) {
10      ...
11    }
12   }
13 }
```

Как и в случае синхронизации метода, код синхронизируемого блока может выполняться только одним потоком на каждый экземпляр `MyObject`. Это означает, что если `thread1` и `thread2` работают с общим экземпляром `MyObject`, в любой момент времени блок кода сможет выполняться только одним из потоков.

Блокировки

Для более тонкой настройки можно воспользоваться блокировкой (lock). Блокировка (монитор) используется для синхронизации доступа к совместно используемому ресурсу, связывая ресурс с блокировкой. Перед получением доступа к совместному ресурсу поток запрашивает блокировку. В любой момент времени только один поток может обладать блокировкой, и поэтому только один поток может получить доступ к совместно используемому ресурсу.

Блокировка чаще всего используется в том случае, если к одному ресурсу существуют обращения сразу из нескольких мест, но *в любой момент времени* с ресурсом должен работать только один поток. Пример:

```
1  public class LockedATM {
2    private Lock lock;
3    private int balance = 100;
4
5    public LockedATM() {
6      lock = new ReentrantLock();
7    }
```

```
8
9    public int withdraw(int value) {
10      lock.lock();
11      int temp = balance;
12      try {
13        Thread.sleep(100);
14        temp = temp - value;
15        Thread.sleep(100);
16        balance = temp;
17      } catch (InterruptedException e) {    }
18      lock.unlock();
19      return temp;
20    }
21
22    public int deposit(int value) {
23      lock.lock();
24      int temp = balance;
25      try {
26        Thread.sleep(100);
27        temp = temp + value;
28        Thread.sleep(300);
29        balance = temp;
30      } catch (InterruptedException e) {    }
31      lock.unlock();
32      return temp;
33    }
34  }
```

Конечно, мы добавили код, преднамеренно замедляющий выполнение `withdraw` и `deposit`, поскольку это помогает продемонстрировать потенциальные проблемы. Возможно, ваш код будет отличаться от этого, но ситуация, которую отражает этот пример, более чем реальна. Использование блокировки поможет защитить совместно используемый ресурс от непредвиденных изменений.

Взаимные блокировки и их предотвращение

Взаимная блокировка — это ситуация, в которой поток блокируется в ожидании ресурса другого потока, а другой поток ожидает освобождения объекта блокировки, удерживаемого первым (или эквивалентная ситуация с несколькими потоками). Поскольку каждый поток находится в ожидании другого, оба остаются в таком состоянии навсегда.

Для возникновения взаимной блокировки должны быть выполнены все четыре условия:

1. **Взаимное исключение**: в любой момент времени ресурс доступен только для одного процесса. (Или, говоря более корректно, — существует ограниченный доступ к ресурсу. Взаимная блокировка также возможна в том случае, если ресурс поддерживает ограниченное количество подключений.)
2. **Удержание и ожидание**: удерживающие ресурс процессы могут запросить дополнительные ресурсы, не освобождая при этом текущие.
3. **Отсутствие вытеснения**: один процесс не может насильственно освободить ресурс другого процесса.

4. **Круговое ожидание**: два или более процесса формируют замкнутую цепочку, в которой каждый процесс ожидает ресурс следующего процесса.

Предотвращение взаимной блокировки можно обеспечить, если устранить любое из перечисленных условий, но этого не так просто добиться. Например, условие 1 удалить сложно, потому что множество ресурсов используют один процесс в один и тот же промежуток времени (например, принтеры). Большинство алгоритмов предотвращения мертвой блокировки чаще всего устраняют условие 4.

Вопросы собеседования

15.1. В чем разница между потоком и процессом?

Подсказки: 405

15.2. Как оценить время, затрачиваемое на переключение контекста?

Подсказки: 403, 407, 415, 441

15.3. В знаменитой задаче об обедающих философах каждый из них имеет только одну палочку для еды. Философу нужно две палочки, и он всегда поднимает левую палочку, а потом — правую. Взаимная блокировка произойдет, если все философы будут одновременно использовать левую палочку. Используя потоки и блокировки, реализуйте модель задачи, предотвращающую взаимные блокировки.

Подсказки: 419, 437

15.4. Разработайте класс, предоставляющий блокировку только в том случае, если это не приведет к созданию взаимной блокировки.

Подсказки: 422, 434

15.5. Имеется следующий код:

```
public class Foo {
  public Foo() {  ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

Один и тот же экземпляр `Foo` передается трем различным потокам: `ThreadA` вызывает `first`, `ThreadB` вызывает `second`, а `threadC` вызывает `third`. Разработайте механизм, гарантирующий, что метод `first` будет вызван перед `second`, а метод `second` будет вызван перед `third`.

Подсказки: 417, 433, 446

15.6. Имеется класс с синхронизированным методом `a` и обычным методом `B`. Если у вас есть два потока в одном экземпляре программы, смогут ли они оба выполнить `a` одновременно? Могут ли они выполнять `a` и `B` одновременно?

Подсказки: 429

15.7. В классической задаче требуется вывести последовательность чисел от 1 до n. Если число кратно 3, то выводится сообщение «Fizz». Если число кратно 5, то выводится сообщение «Buzz». Если число кратно и 3, и 5, выводится сообщение «FizzBuzz». Разработайте многопоточную реализацию этой задачи. Один поток проверяет кратность 3 и выводит «Fizz». Другой поток отвечает за проверку кратности 5 и выводит «Buzz». Третий поток отвечает за проверку кратности 3 и 5 и выводит «FizzBuzz». Четвертый поток работает с числами.

Подсказки: 414, 439, 447, 458

Подсказки начинаются на с. 713 (скачайте ч. XIII на сайте изд-ва «Питер»).

16

Задачи умеренной сложности

16.1. Напишите функцию, которая переставляет значения переменных «на месте» (то есть без использования временных переменных).

Подсказки: 492, 716, 737

16.2. Напишите метод для определения относительной частоты вхождения слов в книге. А если алгоритм должен выполняться многократно?

Подсказки: 489, 536

16.3. Для двух отрезков, заданных начальной и конечной точками, вычислите точку пересечения (если она существует).

Подсказки: 465, 472, 497, 517, 527

16.4. Разработайте алгоритм, проверяющий результат игры в крестики-нолики.

Подсказки: 710, 732

16.5. Напишите алгоритм, вычисляющий число завершающих нулей в `n!`.

Подсказки: 585, 711, 729, 733, 745

16.6. Для двух целочисленных массивов найдите пару значений (по одному значению из каждого массива) с минимальной (неотрицательной) разностью. Верните эту разность.

Пример:

Ввод: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Вывод: 3 для пары (11, 8).

Подсказки: 632, 670, 679

16.7. Напишите метод, находящий максимальное из двух чисел без использования `if-else` или любых других операторов сравнения.

Подсказки: 473, 513, 707, 728

16.8. Задано целое число. Выведите его значение в текстовом виде (например, «одна тысяча двести тридцать четыре»).

Подсказки: 502, 588, 688

16.9. Напишите методы, реализующие операции умножения, вычитания и деления целых чисел. Результатом во всех случаях должно быть целое число. В коде разрешается использовать только оператор сложения.

Подсказки: 572, 600, 613, 648

16.10. Имеется список людей с годами рождения и смерти. Напишите метод для вычисления года, в котором количество живых людей из списка было максимальным. Предполагается, что все люди родились в промежутке от 1900 до 2000 г. (включительно). Если человек прожил хотя бы часть года, он включается в счетчик этого года. Например, человек, родившийся в 1908 г. и умерший в 1909 г., учитывается как в 1908-м, так и в 1909 г.

Подсказки: 476, 490, 507, 514, 523, 532, 541, 549, 576

16.11. Вы строите трамплин для прыжков в воду, складывая деревянные планки концом к концу. Планки делятся на два типа: длины `shorter` и длины `longer`. Необходимо использовать ровно `K` планок. Напишите метод, генерирующий все возможные длины трамплина.

Подсказки: 690, 700, 715, 722, 740, 747

16.12. Разметка XML занимает слишком много места. Закодируйте ее так, чтобы каждый тег заменялся заранее определенным целым значением:

```
Элемент --> Тег Атрибуты END ДочерниеТеги END
Атрибут --> Тег Значение
END --> 0
Тег --> любое предопределенное целое значение
Значение --> строка значение END
```

Для примера преобразуем в сжатую форму следующую разметку XML. Предполагается, что `family -> 1`, `person -> 2`, `firstName -> 3`, `lastName -> 4`, `state -> 5`:

```
<family lastName="McDowell" state="CA">
   <person firstName="Gayle">Some Message</person>
</family>
```

После преобразования код будет иметь вид:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Напишите код, выводящий закодированную версию XML-элемента (переданного в объектах `Element` и `Attributes`).

Подсказки: 466

16.13. Для двух квадратов на плоскости найдите линию, которая делит эти два квадрата пополам. Предполагается, что верхняя и нижняя стороны квадрата проходят параллельно оси x.

Подсказки: 468, 479, 528, 560

16.14. Для заданного набора точек на плоскости найдите линию, проходящую через максимальное количество точек.

Подсказки: 491, 520, 529, 563

16.15. В игру «Великий комбинатор» играют следующим образом:

В каждой из четырех ячеек находится шар красного (`R`), желтого (`Y`), зеленого (`G`) или синего (`B`) цвета. Последовательность `RGGB` означает, что в ячейке 1 — красный шар, в ячейках 2 и 3 — зеленый, в ячейке 4 — синий. Пользователь должен угадать цвета шаров — например, `YRGB`. Если вы угадали правильный цвет в правильной ячейке, то вы получаете «хит». Если вы

назвали цвет, который присутствует в раскладе, но находится в другой ячейке, вы получаете «псевдохит». Ячейка с «хитом» не засчитывается за «псевдохит». Например, если фактический расклад `RGBY`, а ваше предположение `GGRR`, то ответ должен быть: один «хит» и один «псевдохит».

Напишите метод, который возвращает число «хитов» и «псевдохитов».

Подсказки: 639, 730

16.16. Имеется массив целых чисел. Напишите метод для поиска таких индексов `m` и `n`, что для полной сортировки массива достаточно будет отсортировать элементы от `m` до `n`. Минимизируйте `n` и `m` (то есть найдите наименьшую такую последовательность).

Пример:

Ввод: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`
Вывод: `(3, 9)`

Подсказки: 482, 553, 667, 708, 735, 746

16.17. Имеется массив целых чисел (как положительных, так и отрицательных). Найдите непрерывную последовательность с наибольшей суммой. Верните найденную сумму.

Пример:

Ввод: `2, -8, 3, -2, 4, -10`

Вывод: `5 (соответствующая последовательность: 3, -2, 4)`

Подсказки: 531, 551, 567, 594, 614

16.18. Заданы две строки — `pattern` и `value`. Строка `pattern` состоит из букв `a` и `b`, описывающих шаблон чередования подстрок в строке. Например, строка `catcatgocatgo` соответствует шаблону `aabab` (где `cat` представляет `a`, а `go` – `b`). Также строка соответствует таким шаблонам, как `a`, `ab` и `b`.

Напишите метод для определения того, соответствует ли `value` шаблону `pattern`.

Подсказки: 631, 643, 653, 663, 685, 718, 727

16.19. Целочисленная матрица представляет участок земли; значение каждого элемента матрицы обозначает высоту над уровнем моря. Нулевые элементы представляют воду. Назовем «озером» область водных участков, связанных по вертикали, горизонтали или диагонали. Размер озера равен общему количеству соединенных водных участков. Напишите метод для вычисления размеров всех озер в матрице.

Пример:

Ввод:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Вывод: 2, 4, 1 *(в любом порядке)*

Подсказки: 674, 687, 706, 723

16.20. На старых сотовых телефонах при наборе текста пользователь нажимал клавиши на цифровой клавиатуре, а телефон выдавал список слов, соответствующих введенным цифрам. Каждая цифра представляла набор от 0 до 4 цифр. Реализуйте алгоритм для получения списка слов, соответствующих заданной последовательности цифр. Предполагается, что список допустимых слов задан (в любой структуре данных на ваш выбор). Следующая диаграмма поясняет процесс подбора:

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

Пример:

Ввод: `8733`

Вывод: `tree, used`

Подсказки: 471, 487, 654, 703, 726, 744

16.21. Имеются два целочисленных массива. Найдите пару значений (по одному из каждого массива), которые можно поменять местами, чтобы суммы элементов двух массивов были одинаковыми.

Пример:

Ввод: `{4, 1, 2, 1, 1, 2}` и `{3, 6, 3, 3}`

Вывод: `{1, 3}`

Подсказки: 545, 557, 564, 571, 583, 592, 602, 606, 635

16.22. Муравей сидит на бесконечной сетке из белых и черных квадратов. В исходном состоянии он смотрит вправо. На каждом шаге он поступает следующим образом:

(1) На белом квадрате муравей изменяет цвет квадрата, поворачивает на 90 градусов направо (по часовой стрелке) и перемещается вперед на один квадрат.

(2) На черном квадрате муравей изменяет цвет квадрата, поворачивает на 90 градусов налево (против часовой стрелки) и перемещается вперед на один квадрат.

Напишите программу, моделирующую первые K перемещений муравья. Выведите итоговое состояние сетки. Структура данных для представления сетки не задана — вы должны спроектировать ее самостоятельно. Входные данные метода состоят из значения K. Выведите итоговое состояние сетки без возвращения результата. Сигнатура метода должна выглядеть примерно так: `void printKMoves(int K)`.

Подсказки: 474, 481, 533, 540, 559, 570, 599, 616, 627

16.23. Реализуйте метод `rand7()` на базе метода `rand5()`. Другими словами, имеется метод, генерирующий случайные числа в диапазоне от 0 до 4 (включительно). Напишите метод, который использует его для генерирования случайного числа в диапазоне от 0 до 6 (включительно).

Подсказки: 505, 574, 637, 668, 697, 720

16.24. Разработайте алгоритм для поиска в массиве всех пар целых чисел, сумма которых равна заданному значению.

Подсказки: 548, 597, 644, 673

16.25. Разработайте и постройте кэш с вытеснением по давности использования (LRU, Least Recently Used). Кэш должен связывать ключи со значениями (для выполнения операций вставки и получения значения, ассоциированного с заданным ключом) и инициализироваться максимальным размером. При заполнении кэша должен вытесняться элемент, который не использовался дольше всех остальных.

Подсказки: 524, 630, 694

16.26. Вычислите результат заданной арифметической операции, состоящей из положительных целых чисел, а также операторов +, –, * и /.

Пример:

Ввод: `2*3+5/6*3+15`

Вывод: `23.5`

Подсказки: 521, 624, 665, 698

Подсказки начинаются на с. 715 (скачайте ч. XIII на сайте изд-ва «Питер»).

17

Сложные задачи

17.1. Напишите функцию суммирования двух чисел без использования «+» или любых других арифметических операторов.

Подсказки: 467, 544, 601, 628, 642, 664, 692, 712, 724

17.2. Напишите метод, моделирующий тасование карточной колоды. Колода должна быть идеально перетасована — иначе говоря, все 52! перестановки карт должны быть равновероятными. Предполагается, что у вас в распоряжении имеется идеальный генератор случайных чисел.

Подсказки: 483, 579, 634

17.3. Напишите метод, генерирующий случайное множество из `m` целых чисел из массива размером `n`. Все элементы должны выбираться с одинаковой вероятностью.

Подсказки: 494, 596

17.4. Массив `A` содержит все целые числа от 0 до `n`, кроме одного. Считается, что обратиться ко всему целому числу в `A` за одну операцию невозможно. Элементы `A` представлены в двоичной форме, и для работы с ними может использоваться только одна операция — «получить j-й бит элемента `A[i]`», выполняемая за постоянное время. Напишите код для поиска отсутствующего целого числа. Удастся ли вам выполнить поиск за время $O(n)$?

Подсказки: 610, 659, 683

17.5. Имеется массив, заполненный буквами и цифрами. Найдите самый длинный подмассив с равным количеством букв и цифр.

Подсказки: 485, 515, 619, 671, 713

17.6. Напишите метод, который будет подсчитывать количество цифр «2» в записи чисел от 0 до `n` (включительно).

Пример:

Ввод: 25

Вывод: 9 (2, 12, 20, 21, 22, 23, 24 и 25. Обратите внимание: в числе 22 — две двойки).

Подсказки: 573, 612, 641

17.7. Правительство ежегодно публикует список 10 000 самых распространенных имен, выбираемых родителями, и их частоты (количество детей с каждым именем). Единственная проблема заключается в том, что некоторые имена существуют в нескольких вариантах написания. Например, «John»

и «Jon» — фактически одно имя, но в списке они будут занимать две разные позиции. Для двух заданных списков (один содержит имена/частоты, другой — пары эквивалентных имен) напишите алгоритм для вывода нового списка истинной частоты каждого имени. Обратите внимание: если John и Jon являются синонимами и при этом Jon и Johnny являются синонимами, то и John и Johnny являются синонимами (отношение транзитивно и симметрично). В окончательном списке в качестве «настоящего» может использоваться любое имя.

Пример:

Ввод:

Имена: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Синонимы: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Вывод: John (27), Kris (36)

Подсказки: 478, 493, 512, 537, 586, 605, 655, 675, 704

17.8. Цирк готовит новый номер — пирамиду из людей, стоящих на плечах друг у друга. По практическим и эстетическим соображениям люди, стоящие выше, должны быть ниже ростом и легче, чем люди, находящиеся в основании пирамиды. Известен вес и рост каждого акробата; напишите метод, вычисляющий наибольшее возможное число человек в пирамиде.

Пример:

Ввод: `(ht, wt)`: `(65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)`

Вывод: максимальная высота пирамиды — 6 человек, сверху вниз: `(56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)`

Подсказки: 638, 657, 666, 682, 699

17.9. Разработайте алгоритм, позволяющий найти `k`-е число, простыми множителями которого являются только числа 3, 5 и 7. Обратите внимание: 3, 5 и 7 не обязаны быть множителями, но других простых множителей быть не должно. Например, несколько первых таких чисел — 1, 3, 5, 7, 9, 15, 21.

Подсказки: 488, 508, 550, 591, 622, 660, 686

17.10. «Доминирующим значением» называется значение, присутствующее более чем в половине элементов массива. Для заданного массива с положительными целыми числами найдите доминирующее значение. Если доминирующего значения не существует, верните –1. Поиск должен быть выполнен за время $O(N)$ и с затратами памяти $O(1)$.

Пример:

Ввод: `1 2 5 9 5 9 5 5 5`

Вывод: 5

Подсказки: 522, 566, 604, 620, 650

17.11. Имеется большой текстовый файл, содержащий слова. Напишите код, который позволяет найти минимальное расстояние в файле (выражаемое количеством слов) между двумя заданными словами. Если операция будет

многократно выполняться для одного файла (но с разными парами слов), можно ли оптимизировать решение?

Подсказки: 486, 501, 538, 558, 633

17.12. Рассмотрим простую структуру данных `BiNode`, содержащую указатели на два других узла:

```
public class BiNode {
    public BiNode node1, node2;
    public int data;
}
```

Структура данных `BiNode` может использоваться для представления бинарного дерева (где `node1` — левый узел и `node2` — правый узел) или двунаправленного связного списка (где `node1` — предыдущий узел, а `node2` — следующий узел). Реализуйте метод, преобразующий бинарное дерево поиска (реализованное с помощью `BiNode`) в двунаправленный связный список. Значения должны храниться упорядоченно, а операция должна выполняться «на месте» (то есть прямо в исходной структуре данных).

Подсказки: 509, 608, 646, 680, 701, 719

17.13. О нет! Вы только что закончили длинный документ, но из-за неудачной операции поиска/замены из документа пропали все пробелы, знаки препинания и прописные буквы. Предложение `I reset the computer. It still didn't boot!` превратилось в `iresetthecomputeritstilldidntboot`. Со знаками препинания и прописными буквами разберемся позднее, сначала нужно разделить строку на слова. Большинство слов находится в словаре, но есть и исключения. Для заданного словаря (списка строк) и документа (строки) разработайте алгоритм, выполняющий оптимальную деконкатенацию строки. Алгоритм должен минимизировать число нераспознанных последовательностей символов.

Пример:

Ввод: `"jesslookedjustliketimherbrother"`

Вывод: `"JESS looked just like TIM her brother"` (7 нераспознанных символов).

Подсказки: 496, 623, 656, 677, 739, 749

17.14. Разработайте алгоритм для поиска наименьших `K` чисел в массиве.

Подсказки: 470, 530, 552, 593, 625, 647, 661, 678

17.15. Для заданного списка слов напишите алгоритм поиска самого длинного слова, образованного другими словами, входящими в список.

Пример:

Ввод: `cat, banana, dog, nana, walk, walker, dogwalker`

Вывод: `dogwalker`

Подсказки: 475, 499, 543, 589

17.16. Популярная массажистка получает серию заявок на проведение массажа и пытается решить, какие из них следует принять. Между сеансами ей

нужен 15-минутный перерыв, так что она не может принять заявки, следующие непосредственно друг за другом. Для заданной последовательности заявок (продолжительность всех сеансов кратна 15 минутам, перекрытий нет, перемещение невозможно) найдите оптимальный набор (с наибольшей суммарной продолжительностью в минутах), который может быть обслужен массажисткой. Верните суммарную продолжительность в минутах.

Пример:

Ввод: `{30, 15, 60, 75, 45, 15, 15, 45}`

Вывод: `180 минут ({30, 60, 45, 45})`.

Подсказки: 495, 504, 516, 526, 542, 554, 562, 568, 578, 587, 607

17.17. Для заданной строки `b` и массива `T` строк с меньшим размером напишите метод для поиска в `b` всех меньших строк из `T`.

Подсказки: 480, 582, 617, 743

17.18. Даны два массива, короткий (не содержащий одинаковых элементов) и длинный. Найдите в длинном массиве кратчайший подмассив, содержащий все элементы короткого массива. Элементы могут следовать в любом порядке.

Пример:

Ввод: {1, 5, 9} | {7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}

Вывод: [7, 10] (*подчеркнутая часть*)

Подсказки: 645, 652, 669, 681, 691, 725, 731, 741

17.19. Имеется массив, в котором каждое из чисел от 1 до N встречается ровно один раз — кроме одного отсутствующего числа. Как найти отсутствующее число за время $O(N)$ и с затратами памяти $O(1)$? А если отсутствуют два числа?

Подсказки: 503, 590, 609, 626, 649, 672, 689, 696, 702, 717

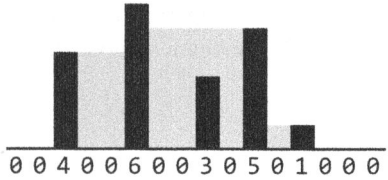
17.20. Числа генерируются случайным образом и передаются методу. Напишите программу для вычисления медианы и ее обновления по мере поступления новых значений.

Подсказки: 519, 546, 575, 709

17.21. Имеется гистограмма (столбцовая диаграмма). Разработайте алгоритм для вычисления объема воды, которая сможет удержаться на гистограмме (см. ниже). Предполагается, что ширина каждого столбца гистограммы равна 1.

Пример (черные полосы — столбцы гистограммы, серые полосы — вода):

Ввод: `{0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}`



Вывод: 26

Подсказки: 629, 640, 651, 658, 662, 676, 693, 734, 742

17.22. Для двух слов одинаковой длины, входящих в словарь, напишите метод последовательного превращения одного слова в другое с изменением одной буквы за один шаг. Каждое новое слово должно присутствовать в словаре.

Пример:

Ввод: DAMP, LIKE

Вывод: DAMP -> LAMP -> LIMP -> LIME -> LIKE

Подсказки: 506, 535, 556, 580, 598, 618, 738

17.23. Имеется квадратная матрица, каждая ячейка (пиксел) которой может быть черной или белой. Разработайте алгоритм поиска максимального субквадрата, у которого все четыре стороны заполнены черными пикселами.

Подсказки: 684, 695, 705, 714, 721, 736

17.24. Для заданной матрицы размером $N \times N$, содержащей положительные и отрицательные числа, напишите код поиска субматрицы с максимально возможной суммой.

Подсказки: 469, 511, 525, 539, 565, 581, 595, 615, 621

17.25. Имеется список из миллионов слов. Разработайте алгоритм, создающий максимально возможный прямоугольник из букв, так чтобы каждая строка и каждый столбец образовывали слово (при чтении слева направо и сверху вниз). Слова могут выбираться в любом порядке, строки должны быть одинаковой длины, а столбцы — одинаковой высоты.

Подсказки: 477, 500, 748

17.26. Степень сходства двух документов (каждый из которых состоит из разных слов) определяется как отношение между размером пересечения и размером объединения. Например, если документы состоят из целых чисел, то степень сходства между `{1, 5, 3}` и `{1, 7, 2, 3}` равна `0.4`, потому что размер пересечения равен 2, а размер объединения равен 5.

Имеется длинный список документов (с разными значениями, с каждым документом связывается идентификатор), для которых степень сходства считается «минималистической». Иначе говоря, для любых двух произвольно выбранных документов степень сходства с большой вероятностью равна 0. Разработайте алгоритм, который возвращает списки пар идентификаторов документов и степеней сходства между ними.

Выведите только те пары, у которых степень сходства больше 0. Пустые документы вообще не включаются в вывод. Для простоты можно считать,

что каждый документ представлен массивом неповторяющихся целых чисел.

Пример:

Ввод:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Вывод:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

Подсказки: 484, 498, 510, 518, 534, 547, 555, 561, 569, 577, 584, 603, 611, 636

Подсказки начинаются на с. 715 (скачайте ч. XIII на сайте изд-ва «Питер»).

Часть X

РЕШЕНИЯ

Присоединяйтесь к нам на www.CrackingTheCodingInterview.com, *просматривайте код решений на других языках программирования, загружайте полные решения, обсуждайте задачи из этой книги с другими пользователями, задавайте вопросы, сообщайте об обнаруженных ошибках, просматривайте списки найденных опечаток и обращайтесь за дополнительными советами.*

1

Массивы и строки

1.1. Реализуйте алгоритм, определяющий, все ли символы в строке встречаются только один раз. А если при этом запрещено использование дополнительных структур данных?

РЕШЕНИЕ

Прежде всего уточните у интервьюера, о какой строке идет речь: обычной ASCII-строке или строке Юникода? Этот вопрос продемонстрирует ваше внимание к деталям и глубокое понимание информатики. Предположим, что используется набор символов ASCII. Если это не так, то нужно будет увеличить объем памяти для хранения строки, но логика решения останется неизменной.

Одно из возможных решений — создание массива логических значений, где флаг с индексом `i` указывает, присутствует ли символ алфавита `i` в строке. При повторном обнаружении этого символа можно сразу возвращать `false`.

Также можно немедленно возвращать `false`, если длина строки превышает количество символов в алфавите. В конце концов, не может существовать строки с 280 уникальными символами, если символов всего 128.

Также можно предположить, что алфавит состоит из 256 символов, как в случае расширенной кодировки ASCII. Не забудьте объяснить свои предположения интервьюеру.

Реализация этого алгоритма может выглядеть примерно так:

```
1  boolean isUniqueChars(String str) {
2    if (str.length() > 128) return false;
3
4    boolean[] char_set = new boolean[128];
5    for (int i = 0; i < str.length(); i++) {
6      int val = str.charAt(i);
7      if (char_set[val]) { // Символ уже встречался в строке
8        return false;
9     }
10      char_set[val] = true;
11    }
12    return true;
13 }
```

Алгоритм характеризуется временной сложностью $O(n)$, где n — длина строки, и пространственной сложностью $O(1)$. (В принципе, также можно считать, что временная сложность равна $O(1)$, так как цикл `for` никогда не будет выполняться более чем для 128 символов.) Если предположение о фиксированном наборе символов

нежелательно, то пространственная сложность может быть выражена в виде $O(c)$, а временная — $O(min(c, n))$ или $O(c)$, где c — размер набора символов.

Затраты памяти можно сократить в 8 раз за счет использования битового вектора. В следующем коде предполагается, что строка содержит только символы в нижнем регистре `a-z`. Это позволит нам ограничиться использованием одного значения `int`.

```
1  boolean isUniqueChars(String str) {
2    int checker = 0;
3    for (int i = 0; i < str.length(); i++) {
4      int val = str.charAt(i) - 'a';
5      if ((checker & (1 << val)) > 0) {
6        return false;
7      }
8      checker |= (1 << val);
9    }
10    return true;
11  }
```

Также возможно действовать иначе:

1. Сравнить каждый символ строки со всеми остальными символами строки. Это потребует $O(n^2)$ времени и $O(1)$ памяти.
2. Если изменение строки разрешено, то можно ее отсортировать (что потребует $O(n \log(n))$ времени), а затем последовательно проверить строку на идентичность соседних символов. Будьте внимательны: некоторые алгоритмы сортировки требуют больших объемов памяти.

Эти решения не оптимальны, но в некоторых ситуациях они могут лучше соответствовать специфике конкретной задачи.

1.2. Для двух строк напишите метод, определяющий, является ли одна строка перестановкой другой.

РЕШЕНИЕ

Как обычно, сразу уточните некоторые подробности у интервьюера. Следует узнать, различается ли регистр символов при сравнении перестановок. Другими словами, является ли строка `"God"` перестановкой `"dog"`? Дополнительно нужно выяснить, учитываются ли пробелы. Будем считать, что в данной задаче регистр символов учитывается, а пробелы являются существенными (соответственно строки `"dog "` и `"dog"` считаются различающимися).

Заметим, что строки разной длины не могут быть перестановками. Существуют два простых способа решить эту задачу, причем в обоих случаях применяется эта оптимизация.

Способ 1. Сортировка строк

Если строки являются перестановками, то они состоят из одинаковых символов, расположенных в разном порядке. Следовательно, сортировка двух строк расставит

две перестановки в одном порядке. Остается только сравнить две отсортированные версии строк.

```
1  String sort(String s) {
2    char[] content = s.toCharArray();
3    java.util.Arrays.sort(content);
4    return new String(content);
5  }
6
7  boolean permutation(String s, String t) {
8    if (s.length() != t.length()) {
9      return false;
10    }
11    return sort(s).equals(sort(t));
12 }
```

Хотя этот алгоритм не является оптимальным в некоторых отношениях, у него есть явное достоинство: он прост, нагляден и логичен. С практической точки зрения это превосходное решение задачи.

Однако если эффективность особенно важна, придется реализовать другой алгоритм.

Способ 2. Проверка счетчиков идентичных символов

Для реализации этого алгоритма можно воспользоваться определением перестановки (два слова с одинаковым количеством вхождений каждого символа). Мы просто перебираем содержимое строк и подсчитываем, сколько раз каждый символ встречается в строке, после чего сравниваем два массива.

```
1  boolean permutation(String s, String t) {
2    if (s.length() != t.length()) {
3      return false;
4    }
5
6    int[] letters = new int[128]; // Предположение
7
8    char[] s_array = s.toCharArray();
9    for (char c : s_array) { // Подсчет вхождений каждого символа в s.
10     letters[c]++;
11   }
12
13   for (int i = 0; i < t.length(); i++) {
14     int c = (int) t.charAt(i);
15     letters[c]--;
16     if (letters[c] < 0) {
17       return false;
18     }
19   }
20
21   return true;
22 }
```

Обратите внимание на предположение в строке 6. На собеседовании вы всегда сможете уточнить у интервьюера размер набора символов. (В данном случае используется набор символов ASCII.)

1.3. Напишите метод, заменяющий все пробелы в строке символами '%20'. Можете считать, что длина строки позволяет сохранить дополнительные символы, а фактическая длина строки известна заранее. (Примечание: при реализации метода на Java для выполнения операции «на месте» используйте символьный массив.)

Пример:

Ввод: `"Mr John Smith    ", 13`

Вывод: `"Mr%20John%20Smith"`

РЕШЕНИЕ

Общий подход к задачам обработки строк — изменять строку, начиная с конца и двигаясь от конца строки к началу. Это полезно, потому что в конце строки есть дополнительное пространство, в котором можно изменять символы, не думая о возможной потере информации.

Воспользуемся этим методом в данной задаче. Алгоритм использует двупроходное сканирование. При первом проходе мы подсчитываем, сколько пробелов находится в строке. Умножая полученное значение на 3, мы получим количество дополнительных символов в результирующей строке. При втором проходе, который осуществляется в обратном направлении, выполняется фактическая модификация строки. Обнаруживая пробел, мы заменяем его на `%20`. Если символ не является пробелом, то он просто копируется.

Реализация данного алгоритма:

```
1   void replaceSpaces(char[] str, int length) {
2    int spaceCount = 0, newLength, i;
3    for (i = 0; i < length; i++) {
4      if (str[i] == ' ') {
5        spaceCount++;
6     }
7    }
8    newLength = length + spaceCount * 2;
9    str[newLength] = '\0';
10    for (i = length - 1; i >= 0; i--) {
11      if (str[i] == ' ') {
12        str[newLength - 1] = '0';
13        str[newLength - 2] = '2';
14        str[newLength - 3] = '%';
15        newLength = newLength - 3;
16      } else {
17        str[newLength - 1] = str[i];
18        newLength = newLength - 1;
19     }
20    }
21  }
```

В решении этой задачи мы использовали символьные массивы, так как строки в Java являются неизменными. Если использовать непосредственно строки, функции придется возвращать новую копию строки, но задача будет решаться за один проход.

1.4. Напишите функцию, которая проверяет, является ли заданная строка перестановкой палиндрома. (Палиндром — слово или фраза, одинаково читающиеся в прямом и обратном направлении; перестановка — строка, содержащая те же символы в другом порядке.) Палиндром не ограничивается словами из словаря.

Пример:

Ввод: `Tact Coa`

Вывод: `True (перестановки: "taco cat", "atco cta", и т. д.)`

РЕШЕНИЕ

При решении этой задачи полезно задаться вопросом, какими характеристиками должна обладать строка, являющаяся перестановкой палиндрома, то есть по сути спросить, какими «определяющими характеристиками» должна обладать такая строка.

Палиндром — строка, читающаяся одинаково в прямом и обратном направлении. Следовательно, чтобы решить, является ли строка перестановкой палиндрома, нужно проверить, возможно ли записать ее так, чтобы она одинаково читалась в прямом и обратном направлении.

Что необходимо для того, чтобы набор символов можно было записать одинаково в прямом и обратном направлении? Почти все символы должны входить в строку в четном количестве (две одинаковые половины, располагающиеся симметрично от центра). Максимум один символ (средний) может входить в строку в нечетном количестве.

Например, мы знаем, что `tactcoapapa` является перестановкой палиндрома, потому что строка содержит два вхождения T, четыре вхождения A, два C, два P и одно вхождение O. Буква O занимает центральную позицию во всех возможных палиндромах.

Говоря точнее, в строках равной длины (после удаления всех небуквенных символов) количества вхождений всех символов должны быть четными. Конечно, строка четной длины не может содержать нечетное количество вхождений ровно одного символа, иначе ее длина не была бы четной (нечетное число + множество четных чисел = нечетное число). Аналогичным образом в строке нечетной длины все количества символов не могут быть четными (сумма четных чисел также четна). Следовательно, чтобы строка была перестановкой палиндрома, достаточно, чтобы в строке было не более одного символа с нечетным счетчиком вхождений. Это правило истинно для строк как четной, так и нечетной длины.

Так мы приходим к первому алгоритму.

Решение 1

Реализация выглядит достаточно тривиально. Алгоритм использует хеш-таблицу для подсчета вхождений каждого символа, после чего перебирает хеш-таблицу и проверяет, что ни один символ не входит в строку в нечетном количестве.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int[] table = buildCharFrequencyTable(phrase);
3    return checkMaxOneOdd(table);
4  }
```

```
5
6  /* Проверяем, что символов с нечетным количеством вхождений не более 1.*/
7  boolean checkMaxOneOdd(int[] table) {
8    boolean foundOdd = false;
9    for (int count : table) {
10     if (count % 2 == 1) {
11       if (foundOdd) {
12         return false;
13     }
14       foundOdd = true;
15    }
16   }
17   return true;
18 }
19
20 /* Каждый символ связывается с числом: a -> 0, b -> 1, c -> 2, и т.д.
21  * (без учета регистра). Небуквенным символам соответствует -1. */
22 int getCharNumber(Character c) {
23   int a = Character.getNumericValue('a');
24   int z = Character.getNumericValue('z');
25   int A = Character.getNumericValue('A');
26   int Z = Character.getNumericValue('Z');
27
28   int val = Character.getNumericValue(c);
29   if (a <= val && val <= z) {
30     return val - a;
31   } else if (A <= val && val <= Z) {
32     return val - A;
33   }
34   return -1;
35 }
36
37 /* Подсчет количества вхождений каждого символа. */
38 int[] buildCharFrequencyTable(String phrase) {
39   int[] table = new int[Character.getNumericValue('z') -
40                         Character.getNumericValue('a') + 1];
41   for (char c : phrase.toCharArray()) {
42     int x = getCharNumber(c);
43     if (x != -1) {
44       table[x]++;
45    }
46   }
47   return table;
48 }
```

Алгоритм выполняется за время $O(N)$, где N — длина строки.

Решение 2

Сократить время $O(N)$ не удастся, так как любому алгоритму всегда придется просматривать всю строку. Тем не менее в алгоритм можно внести несколько второстепенных улучшений. Задача относительно простая, поэтому стоит рассмотреть несколько возможных оптимизаций (или, по крайней мере, улучшений).

Вместо того чтобы проверять количество нечетных вхождений в конце, можно осуществлять проверку в процессе обработки. В тот момент, когда обработка дойдет до конца, ответ будет готов.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int countOdd = 0;
3    int[] table = new int[Character.getNumericValue('z') -
4                          Character.getNumericValue('a') + 1];
5    for (char c : phrase.toCharArray()) {
6      int x = getCharNumber(c);
7      if (x != -1) {
8        table[x]++;
9        if (table[x] % 2 == 1) {
10         countOdd++;
11       } else {
12         countOdd--;
13     }
14     }
15   }
16   return countOdd <= 1;
17 }
```

Здесь необходимо очень четко сказать, что такое решение не обязательно является более эффективным. Оно обладает той же временной сложностью, а иногда может оказаться чуть более медленным. Мы исключили завершающий перебор хеш-таблицы, но теперь для каждого символа в строке выполняются несколько дополнительных строк кода.

Обсудите это решение со своим интервьюером в качестве альтернативы — не обязательно более эффективной.

Решение 3

Если поглубже поразмыслить над задачей, можно заметить, что фактические значения счетчиков не нужны: достаточно знать, является значение каждого счетчика четным или нечетным. Представьте, что перед вами аналог выключателя: если свет выключен, мы не знаем, сколько раз он включался и выключался, но можем быть уверены в том, что количество переключений было четным.

С учетом этого обстоятельства для хранения данных можно использовать одно целое число (как битовый вектор). Обнаружив букву при переборе символов, мы связываем ее с целым числом в диапазоне от 0 до 26 (для английского алфавита), после чего переключаем бит в соответствующей позиции. В конце перебора остается проверить, что не более чем один бит в целом значении содержит 1.

Проверить, что в числе нет ни одного единичного бита, проще простого: достаточно сравнить число с 0. Однако существует очень элегантный способ проверить, что в числе ровно один бит содержит 1.

Представьте целое число вида 00010000. Конечно, можно многократно выполнить сдвиг и убедиться в том, что 1 присутствует только в одном разряде. Есть и другой способ: вычитая 1 из этого числа, мы получим 00001111. Обратите внимание на то, что между этими числами нет перекрытия (в отличие, скажем, от 00101000 — при уменьшении этого числа на 1 получается 00100111).

Следовательно, чтобы проверить, содержит ли число ровно один единичный бит, можно вычесть из него 1 и объединить с новым числом операцией AND — результат должен быть равен 0.

```
00010000 - 1 = 00001111
00010000 & 00001111 = 0
```

Так мы приходим к окончательной реализации.

```
1  boolean isPermutationOfPalindrome(String phrase) {
2    int bitVector = createBitVector(phrase);
3    return bitVector == 0 || checkExactlyOneBitSet(bitVector);
4  }
5
6  /* Создание битового вектора для строки. Для каждой буквы
7   * со значением i изменить состояние i-го бита. */
8  int createBitVector(String phrase) {
9    int bitVector = 0;
10   for (char c : phrase.toCharArray()) {
11     int x = getCharNumber(c);
12     bitVector = toggle(bitVector, x);
13   }
14   return bitVector;
15 }
16
17 /* Переключение i-го бита в целом значении. */
18 int toggle(int bitVector, int index) {
19   if (index < 0) return bitVector;
20
21   int mask = 1 << index;
22   if ((bitVector & mask) == 0) {
23     bitVector |= mask;
24   } else {
25     bitVector &= ~mask;
26   }
27   return bitVector;
28 }
29
30 /* Чтобы проверить, что в числе установлен ровно один бит,
31  * следует вычесть 1 и объединить с исходным числом операцией AND. */
32 boolean checkExactlyOneBitSet(int bitVector) {
33   return (bitVector & (bitVector - 1)) == 0;
34 }
```

Как и другие решения, оно выполняется за время $O(N)$.

Также стоит упомянуть о подходе, который не был исследован в ходе решения. Мы сознательно держались подальше от решений, построенных по принципу «сгенерировать все возможные перестановки и проверить, есть ли среди них палиндромы». Такое решение теоретически работает, но на практике оно совершенно неприемлемо. Генерирование всех перестановок требует факториального времени (которое даже хуже экспоненциального), и для строк длиннее 10–15 символов оно становится нереальным.

Я упоминаю об этом (нереальном) решении только потому, что многие кандидаты, столкнувшись с подобной задачей, говорят: «Чтобы проверить, входит ли A в группу

B, нужно сгенерировать полное содержимое B и проверить, не равен ли один из элементов A». Это не всегда так, о чем наглядно свидетельствует эта задача. Не нужно генерировать все перестановки, чтобы проверить, является ли одна из них палиндромом.

1.5. Существуют три вида модифицирующих операций со строками: вставка символа, удаление символа и замена символа. Напишите функцию, которая проверяет, находятся ли две строки на расстоянии одной модификации (или нуля модификаций).

Пример:

```
pale,  ple  -> true
pales, pale -> true
pale,  bale -> true
pale,  bake -> false
```

РЕШЕНИЕ

Для решения этой задачи можно действовать методом «грубой силы»: чтобы проверить все возможные строки, находящиеся на расстоянии одной модификации, можно проверить результаты удаления каждого символа (и сравнить), проверить результат замены каждого символа (и сравнить) и проверить результат вставки каждого возможного символа (и сравнить). Такой «алгоритм» будет работать слишком медленно, так что мы не будем возиться с его реализацией.

Это одна из тех задач, в которых полезно задуматься над «смыслом» каждой такой операции. Что означает, что две строки находятся на расстоянии одной операции вставки, замены или удаления друг от друга?

- **Замена**: рассмотрим две строки, находящиеся на расстоянии одной замены, — например, «bale» и «pale». Да, это означает, что заменой одного символа можно превратить «bale» в «pale». Но если говорить точнее, это означает, что строки отличаются только в одной позиции.
- **Вставка**: строки «apple» и «aple» находятся на расстоянии одной вставки. Это означает, что при сравнении строки будут идентичными — за исключением сдвига в некоторой позиции строки.
- **Удаление**: строки «apple» и «aple» тоже находятся на расстоянии одного удаления, поскольку операция удаления является обратной по отношению к вставке.

Теперь можно переходить к реализации алгоритма. Вставка и удаление объединяются в один шаг, а операция замены обрабатывается отдельно.

Заметим, что проверять строки на все типы операций (вставка, удаление и замена) не нужно. Длины строк укажут, какую из операций следует проверять.

```
1  boolean oneEditAway(String first, String second) {
2    if (first.length() == second.length()) {
3      return oneEditReplace(first, second);
4    } else if (first.length() + 1 == second.length()) {
5      return oneEditInsert(first, second);
6    } else if (first.length() - 1 == second.length()) {
7      return oneEditInsert(second, first);
```

```
8    }
9    return false;
10 }
11
12 boolean oneEditReplace(String s1, String s2) {
13   boolean foundDifference = false;
14   for (int i = 0; i < s1.length(); i++) {
15     if (s1.charAt(i) != s2.charAt(i)) {
16       if (foundDifference) {
17         return false;
18       }
19
20       foundDifference = true;
21     }
22   }
23   return true;
24 }
25
26 /* Проверить, возможно ли превратить s1 в s2 вставкой символа. */
27 boolean oneEditInsert(String s1, String s2) {
28   int index1 = 0;
29   int index2 = 0;
30   while (index2 < s2.length() && index1 < s1.length()) {
31     if (s1.charAt(index1) != s2.charAt(index2)) {
32       if (index1 != index2) {
33         return false;
34       }
35       index2++;
36     } else {
37       index1++;
38       index2++;
39     }
40   }
41   return true;
42 }
```

Этот алгоритм (как и почти любой другой разумный алгоритм) выполняется за время $O(n)$, где n — длина более короткой строки.

Почему время выполнения определяется более короткой, а не более длинной строкой? Если строки имеют одинаковую длину (плюс-минус один символ), неважно, какая из двух строк будет использоваться для оценки времени выполнения. Если строки сильно различаются по длине, то алгоритм завершится за время O(1). Даже если одна строка окажется очень длинной, это не приведет к заметному увеличению времени выполнения; оно возрастает только в том случае, если длинными будут обе строки.

Вероятно, вы заметили, что код `oneEditReplace` очень похож на код `oneEditInsert`. Их можно объединить в один метод.

Для этого заметим, что оба метода следуют единой логике: программа сравнивает все символы и проверяет, что строки отличаются только в одной позиции. Различаются способы обработки этих различий. Метод `oneEditReplace` просто отмечает обнаруженное различие, тогда как `oneEditInsert` смещает указатель в более длинной строке. Обе ситуации можно обработать в одном методе.

```
boolean oneEditAway(String first, String second) {
  /* Проверка длины. */
  if (Math.abs(first.length() - second.length()) > 1) {
    return false;
  }

  /* Получение более короткой и более длинной строки.*/
  String s1 = first.length() < second.length() ? first : second;
  String s2 = first.length() < second.length() ? second : first;

   int index1 = 0;
   int index2 = 0;
   boolean foundDifference = false;
   while (index2 < s2.length() && index1 < s1.length()) {
     if (s1.charAt(index1) != s2.charAt(index2)) {
       /* Убедиться в том, что это первое найденное различие.*/
       if (foundDifference) return false;
       foundDifference = true;

       if (s1.length() == s2.length()) { // При замене сместить указатель
         index1++;                       // короткой строки.
     }
     } else {
       index1++; // При совпадении сместить указатель короткой строки.
    }
     index2++; // Всегда смещать указатель длинной строки.
  }
  return true;
}
```

Кто-то предпочтет первое решение, потому что оно проще и понятнее. Другие скажут, что второе решение лучше, так как оно более компактно и не содержит дублирующегося кода (который может усложнить сопровождение).

Вы не обязаны становиться на одну из двух сторон. Обсудите достоинства и недостатки каждого решения со своим интервьюером.

1.6. Реализуйте метод для выполнения простейшего сжатия строк с использованием счетчика повторяющихся символов. Например, строка aabcccccaaa превращается в a2b1c5a3. Если «сжатая» строка не становится короче исходной, то метод возвращает исходную строку. Предполагается, что строка состоит только из букв верхнего и нижнего регистра (a—z).

РЕШЕНИЕ

На первый взгляд реализация этого метода выглядит довольно прямолинейно. Мы перебираем содержимое строки, копируя символы в новую строку и подсчитывая повторы. При каждой итерации алгоритм проверяет, совпадает ли текущий символ со следующим. Если символы не совпадают, то сжатая версия добавляется в результат.

С какой сложностью будет выполняться этот код?

```
String compressBad(String str) {
  String compressedString = "";
  int countConsecutive = 0;
  for (int i = 0; i < str.length(); i++) {
```

```
5      countConsecutive++;
6      /* Если следующий символ отличается от текущего, присоединить
7         текущий символ к результату.*/
8      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
9        compressedString += "" + str.charAt(i) + countConsecutive;
10       countConsecutive = 0;
11     }
12    }
13    return compressedString.length() < str.length() ? compressedString : str;
14  }
```

Такое решение работает, но можно ли назвать его эффективным?

Код выполняется за время $O(p + k^2)$, где p — размер исходной строки, а k — количество последовательностей символов. Например, строка `aabccdeeaa` содержит 6 последовательностей символов. Алгоритм выполняется медленно, поскольку в нем используется конкатенация строк, требующая времени $O(n^2)$ (см. `StringBuffer` в гл. 1).

Для улучшения кода можно воспользоваться `StringBuffer`:

```
1  String compress(String str) {
2    StringBuilder compressed = new StringBuilder();
3    int countConsecutive = 0;
4    for (int i = 0; i < str.length(); i++) {
5      countConsecutive++;
6      /* Если следующий символ отличается от текущего, присоединить
7         текущий символ к результату.*/
8      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
9        compressed.append(str.charAt(i));
10       compressed.append(countConsecutive);
11       countConsecutive = 0;
12     }
13    }
14    return compressed.length() < str.length() ? compressed.toString() : str;
15  }
```

Оба решения сначала создают сжатую строку, а затем возвращают более короткую из двух строк: входной и сжатой.

Также можно выполнить проверку заранее — такое решение будет более эффективным в тех случаях, когда строка не содержит множества повторяющихся символов. Таким образом мы избежим создания строки, которая никогда не будет использоваться. С другой стороны, решение требует второго прохода по символам в цикле и появления практически дублирующегося кода.

```
1  String compress(String str) {
2  /* Проверка итоговой длины и возвращение входной строки, если она длиннее. */
3    int finalLength = countCompression(str);
4    if (finalLength >= str.length()) return str;
5
6    StringBuilder compressed = new StringBuilder(finalLength); // Исходная емкость
7    int countConsecutive = 0;
8    for (int i = 0; i < str.length(); i++) {
9      countConsecutive++;
10     /* Если следующий символ отличается от текущего, присоединить
11        текущий символ к результату.*/
```

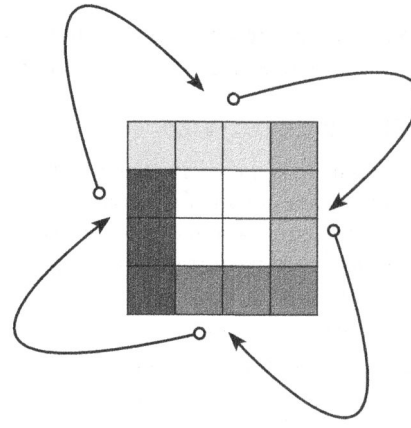
```
12      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
13        compressed.append(str.charAt(i));
14        compressed.append(countConsecutive);
15        countConsecutive = 0;
16      }
17    }
18    return compressed.toString();
19  }
20
21  int countCompression(String str) {
22    int compressedLength = 0;
23    int countConsecutive = 0;
24    for (int i = 0; i < str.length(); i++) {
25      countConsecutive++;
26
27      /* Если следующий символ отличается от текущего, увеличить длину.*/
28      if (i + 1 >= str.length() || str.charAt(i) != str.charAt(i + 1)) {
29        compressedLength += 1 + String.valueOf(countConsecutive).length();
30        countConsecutive = 0;
31      }
32    }
33    return compressedLength;
34  }
```

Другое преимущество такого подхода заключается в том, что экземпляр `StringBuilder` можно заранее инициализировать необходимой емкостью. Без этого `StringBuilder` будет (автоматически) удваивать емкость каждый раз при нехватке места. В итоге емкость может оказаться вдвое больше реально необходимой.

1.7. Имеется изображение, представленное матрицей $N \times N$; каждый пиксел представлен 4 байтами. Напишите метод для поворота изображения на 90 градусов. Удастся ли вам выполнить эту операцию «на месте»?

РЕШЕНИЕ

Поскольку матрица будет поворачиваться на 90°, проще всего использовать повороты по слоям. Мы произведем циклический поворот слоя, двигаясь по кругу: верхнюю сторону на правую, правую — на нижнюю и т. д.



Как поменять местами четыре стороны? Один из вариантов — скопировать верхнюю сторону в массив, затем переместить левую сторону на место верхней, нижнюю — на место левой и т. д. Такое решение потребует памяти $O(N)$, но на самом деле это излишне.

То же самое можно сделать иначе: реализовать последовательную замену элемент за элементом. В этом случае происходит следующее:

```
for i = 0 to n
  temp = top[i];
  top[i] = left[i]
  left[i] = bottom[i]
  bottom[i] = right[i]
  right[i] = temp
```

Замена выполняется на каждом слое, начиная с внешнего, с продвижением вовнутрь. (Также можно действовать в обратном направлении — двигаться от внутреннего слоя к внешнему.)

Код этого алгоритма:

```
void rotate(int[][] matrix, int n) {
  for (int layer = 0; layer < n / 2; ++layer) {
    int first = layer;
    int last = n - 1 - layer;
    for(int i = first; i < last; ++i) {
      int offset = i - first;
      // Сохранить верхнюю сторону
      int top = matrix[first][i];

       // левая сторона -> верхняя сторона
       matrix[first][i] = matrix[last-offset][first];

       // нижняя сторона -> левая сторона
       matrix[last-offset][first] = matrix[last][last - offset];

       // правая сторона -> нижняя сторона
       matrix[last][last - offset] = matrix[i][last];

       // верхняя сторона -> правая сторона
       matrix[i][last] = top;
    }
  }
}
```

Этот алгоритм требует времени $O(N^2)$ — и на лучшее рассчитывать не приходится, поскольку любой алгоритм должен обратиться ко всем N^2 элементам.

1.8. Напишите алгоритм, реализующий следующее условие: если элемент матрицы $M \times N$ равен 0, то весь столбец и вся строка обнуляются.

РЕШЕНИЕ

На первый взгляд задача очень проста — просто пройтись по матрице и для каждого нулевого элемента обнулить соответствующие строку и столбец. Но у такого решения есть один большой недостаток: на очередном шаге мы столкнемся с нулями, которые сами же установили. Невозможно будет понять, установили эти нули мы сами или они присутствовали в матрице изначально. Довольно скоро вся матрица заполнится нулями.

Один из способов — создать вторую матрицу, содержащую флаги «исходных» нулей. Но тогда потребуется сделать два прохода по матрице, что потребует затрат памяти $O(MN)$.

Так ли нам нужно $O(MN)$? Нет. Так как мы собираемся обнулить всю строку и весь столбец, нет необходимости запоминать, что нуль содержался именно в `cell[2][4]`. Достаточно знать, что где-то в строке 2 содержался 0 и в столбце 4 где-то содержался 0. Вся строка и столбец все равно будут обнулены, зачем запоминать точное местоположение нуля?

Ниже приведена реализация этого алгоритма. Мы используем два массива, чтобы отследить все строчки и столбцы с нулями. Затем строки и столбцы обнуляются на основании значений из созданного массива.

```
1  void setZeros(int[][] matrix) {
2    boolean[] row = new boolean[matrix.length];
3    boolean[] column = new boolean[matrix[0].length];
4
5    // Сохранение индексов строки и столбца, содержащих 0
6    for (int i = 0; i < matrix.length; i++) {
7      for (int j = 0; j < matrix[0].length;j++) {
8        if (matrix[i][j] == 0) {
9          row[i] = true;
10         column[j] = true;
11        }
12     }
13   }
14
15   // Обнуление строк
16   for (int i = 0; i < row.length; i++) {
17     if (row[i]) nullifyRow(matrix, i);
18   }
19
20   // Обнуление столбцов
21   for (int j = 0; j < column.length; j++) {
22     if (column[j]) nullifyColumn(matrix, j);
23   }
24 }
25
26 void nullifyRow(int[][] matrix, int row) {
27   for (int j = 0; j < matrix[0].length; j++) {
28     matrix[row][j] = 0;
29   }
30 }
31
32 void nullifyColumn(int[][] matrix, int col) {
33   for (int i = 0; i < matrix.length; i++) {
34     matrix[i][col] = 0;
35   }
36 }
```

Чтобы повысить эффективность алгоритма, можно использовать битовый вектор вместо булевого массива. Затраты времени все равно составят $O(N)$.

Пространственную сложность можно сократить до $O(1)$, используя первую строку как замену для массива `row`, а первый столбец — как замену для массива `column`. Это делается так:

1. Проверить, присутствуют ли нули в первой строке и первом столбце, и задать переменные `rowHasZero` и `columnHasZero`. (Первая строка и первый столбец будут обнулены позднее, если потребуется.)

2. Перебрать остальные элементы матрицы и присвоить `matrix[i][0]` и `matrix[0][j]` нули для всех нулевых элементов `matrix[i][j]`.
3. Перебрать остальные элементы матрицы и обнулить строку `i`, если `matrix[i][0]` содержит 0.
4. Перебрать остальные матрицы и обнулить столбец `j`, если `matrix[0][j]` содержит 0.
5. Обнулить первую строку и первый столбец, если это необходимо (по значениям с шага 1).

Реализация выглядит так:

```
1  void setZeros(int[][] matrix) {
2    boolean rowHasZero = false;
3    boolean colHasZero = false;
4
5    // Проверить, содержит ли первая строка 0.
6    for (int j = 0; j < matrix[0].length; j++) {
7      if (matrix[0][j] == 0) {
8        rowHasZero = true;
9        break;
10     }
11   }
12
13   // Проверить, содержит ли первый столбец 0.
14   for (int i = 0; i < matrix.length; i++) {
15     if (matrix[i][0] == 0) {
16       colHasZero = true;
17       break;
18     }
19   }
20
21   // Проверить нули в остальных элементах массива
22   for (int i = 1; i < matrix.length; i++) {
23     for (int j = 1; j < matrix[0].length;j++) {
24       if (matrix[i][j] == 0) {
25         matrix[i][0] = 0;
26         matrix[0][j] = 0;
27        }
28     }
29   }
30
31   // Обнулить строки на основании значений из первого столбца
32   for (int i = 1; i < matrix.length; i++) {
33     if (matrix[i][0] == 0) {
34       nullifyRow(matrix, i);
35     }
36   }
37
38   // Обнулить столбцы на основании значений из первого столбца
39   for (int j = 1; j < matrix[0].length; j++) {
40     if (matrix[0][j] == 0) {
41       nullifyColumn(matrix, j);
42     }
43   }
44
45   // Обнулить первую строку
```

```
46    if (rowHasZero) {
47      nullifyRow(matrix, 0);
48    }
49
50    // Обнулить первый столбец
51    if (colHasZero) {
52      nullifyColumn(matrix, 0);
53    }
54  }
```

В этом коде часто встречаются действия по принципу «выполнить для строк, затем проделать то же для столбца». В ходе собеседования этот код можно сократить, добавив комментарии и метки TODO с объяснением того, что следующий фрагмент аналогичен предыдущему, но в нем используются строки вместо столбцов. Это позволит вам сосредоточиться на более важных частях алгоритма.

1.9. Допустим, что существует метод `isSubstring`, проверяющий, является ли одно слово подстрокой другого. Для двух строк `s1` и `s2` напишите код, который проверяет, получена ли строка `s2` циклическим сдвигом `s1`, используя только один вызов метода `isSubstring` (пример: слово `waterbottle` получено циклическим сдвигом `erbottlewat`).

РЕШЕНИЕ

Если представить, что `s2` является циклическим сдвигом `s1`, то можно найти точку «поворота». Например, если сдвинуть `waterbottle` после `wat`, будет получена строка `erbottlewat`. При сдвиге `s1` делится на две части: `x` и `y`, которые в другом порядке сохраняются в `s2`.

```
s1 = xy = waterbottle
x = wat
y = erbottle
s2 = yx = erbottlewat
```

Таким образом, необходимо проверить, существует ли вариант разбиения `s1` на `x` и `y`, такой, что `xy=s1`, а `yx=s2`. Независимо от точки разделения `x` и `y`, `yx` всегда будет подстрокой `xyxy`. Таким образом, `s2` всегда будет подстрокой `s1s1`.

Именно так и решается задача: достаточно вызвать функцию `isSubstring(s1s1, s2)`.

Следующий код реализует описанный алгоритм:

```
1  boolean isRotation(String s1, String s2) {
2    int len = s1.length();
3    /* Проверить, что у s1 и s2 одинаковая длина и они не пусты */
4    if (len == s2.length() && len > 0) {
5      /* Конкатенация s1 с s1 в новом буфере */
6      String s1s1 = s1 + s1;
7      return isSubstring(s1s1, s2);
8    }
9    return false;
10 }
```

Время выполнения зависит от времени выполнения `isSubstring`. Но если считать, что `isSubstring` выполняется за время $O(A+B)$ (для строк длины A и B), время выполнения `isRotation` составит $O(N)$.

2

Связные списки

2.1. Напишите код для удаления дубликатов из несортированного связного списка.

Дополнительно

Как вы будете решать задачу, если использовать временный буфер запрещено?

РЕШЕНИЕ

Чтобы удалить дубликаты из связного списка, их нужно сначала найти. Для этого подойдет простая хеш-таблица. В приведенном далее решении перебирается содержимое списка, и каждый элемент добавляется в хеш-таблицу. При обнаружении дубликата он удаляется, и цикл продолжается. За счет использования связного списка всю задачу можно решить за один проход.

```
1  void deleteDups(LinkedListNode n) {
2    HashSet<Integer> set = new HashSet<Integer>();
3    LinkedListNode previous = null;
4    while (n != null) {
5      if (set.contains(n.data)) {
6        previous.next = n.next;
7      } else {
8        set.add(n.data);
9        previous = n;
10     }
11      n = n.next;
12   }
13 }
```

Приведенное решение потребует времени $O(N)$, где N — количество элементов в связном списке.

Дополнительное ограничение: использование буфера запрещено

В этом случае для перебора можно воспользоваться двумя указателями: `current` (перебор связного списка) и `runner` (проверка всех последующих узлов на наличие дубликатов).

```
1  void deleteDups(LinkedListNode head) {
2    LinkedListNode current = head;
3    while (current != null) {
4      /* Удаление всех последующих узлов с тем же значением */
5      LinkedListNode runner = current;
6      while (runner.next != null) {
7        if (runner.next.data == current.data) {
8          runner.next = runner.next.next;
```

```
9         } else {
10           runner = runner.next;
11        }
12      }
13       current = current.next;
14    }
15  }
```

Затраты памяти в этом случае составляют всего $O(1)$, но зато время выполнения увеличивается до $O(N^2)$.

2.2. Реализуйте алгоритм для нахождения в односвязном списке k-го элемента с конца.

РЕШЕНИЕ

Данную задачу можно решить как с применением рекурсии, так и без нее. Вспомните, что рекурсивные решения обычно более понятны, но менее эффективны. Например, рекурсивная реализация этой задачи почти в два раза короче нерекурсивной, но требует затрат памяти $O(n)$, где n — количество элементов связного списка.

При решении данной задачи помните, что мы определили k так, что при передаче k = 1 возвращается последний элемент, k = 2 — предпоследний и т. д. С таким же успехом можно было определить k так, чтобы k = 0 соответствовало последнему элементу.

Решение 1. Размер связного списка известен

Если размер связного списка известен, k-й элемент с конца легко вычислить (длина – k). Достаточно перебрать элементы списка и найти этот элемент. Поскольку такое решение тривиально, то, скорее всего, это совсем не то, чего ожидает интервьюер.

Решение 2. Рекурсивное решение

Такой алгоритм основан на рекурсивном переборе связного списка. По достижении последнего элемента алгоритм начинает обратный отсчет, и счетчик сбрасывается в 0. При каждом шаге счетчик увеличивается на 1. Когда счетчик достигнет k, мы знаем, что достигнут k-й элемент с конца.

Алгоритм реализуется просто и удобно — при условии, что в нашем распоряжении имеется механизм «возврата» целого значения через стек. К сожалению, обычная команда `return` не может вернуть узел вместе со счетчиком. Так как же обойти эту трудность?

Подход А. Не возвращайте элемент

Можно слегка изменить формулировку задачи и ограничиться простым выводом k-го элемента с конца. В этом случае команда `return` просто возвращает значение счетчика.

```
1  int printKthToLast(LinkedListNode head, int k) {
2    if (head == null) {
3      return 0;
4    }
5    int index = printKthToLast(head.next, k) + 1;
```

```
6    if (index == k) {
7      System.out.println(k + "th to last node is " + head.data);
8    }
9    return index;
10 }
```

Конечно, данное решение может считаться действительным только в том случае, если интервьюер с этим согласится.

Подход Б. Используйте C++

Второй способ — использование C++ и передача значения по ссылке. Такой подход позволяет не только вернуть значение узла, но и обновить счетчик.

```
1  node* nthToLast(node* head, int k, int& i) {
2    if (head == NULL) {
3      return NULL;
4    }
5    node* nd = nthToLast(head->next, k, i);
6    i = i + 1;
7    if (i == k) {
8      return head;
9    }
10    return nd;
11 }
12
13 node* nthToLast(node* head, int k) {
14   int i = 0;
15   return nthToLast(head, k, i);
16 }
```

Подход В. Создайте обертку

Как упоминалось ранее, проблема заключается в том, что мы не можем одновременно вернуть счетчик и индекс. Если «обернуть» значение `counter` в простой класс (или простой элемент массива), это позволит имитировать передачу по ссылке.

```
1  class Index {
2    public int value = 0;
3  }
4
5  LinkedListNode kthToLast(LinkedListNode head, int k) {
6    Index idx = new Index();
7    return kthToLast(head, k, idx);
8  }
9
10 LinkedListNode kthToLast(LinkedListNode head, int k, Index idx) {
11   if (head == null) {
12     return null;
13   }
14   LinkedListNode node = kthToLast(head.next, k, idx);
15   idx.value = idx.value + 1;
16   if (idx.value == k) {
17     return head;
18   }
19   return node;
20 }
```

Каждое из представленных решений занимает память $O(n)$ из-за рекурсивных вызовов.

Существует множество других решений, которые здесь не рассматриваются. Можно сохранить счетчик в статической переменной. Или создать класс, в полях которого хранится узел и счетчик, и возвращать экземпляр класса. Независимо от выбранного решения нам нужен способ обновить узел и счетчик так, чтобы они были видны на всех уровнях рекурсивного стека.

Решение 3. Итеративное решение

Итеративное решение получается более сложным, но и более эффективным. Мы используем два указателя, `p1` и `p2`. Они будут разнесены на `k` узлов связного списка; для этого `p2` устанавливается в начало списка, а `p1` перемещается на `k` узлов. Затем оба указателя перемещаются вперед в одном темпе, и `p1` дойдет до конца связного списка через `длина - k` шагов. В этот момент указатель `p2` будет смещен на `длина - k` шагов в списке, то есть на `k` узлов от конца.

Реализация этого алгоритма выглядит так:

```
1  LinkedListNode nthToLast(LinkedListNode head, int k) {
2    LinkedListNode p1 = head;
3    LinkedListNode p2 = head;
4
5    /* Перемещение p1 на k узлов по списку.*/
6    for (int i = 0; i < k; i++) {
7      if (p1 == null) return null; // Выход за границу
8      p1 = p1.next;
9    }
10
11    /* Теперь p1 и p2 перемещаются с одной скоростью. Когда p1 дойдет
12     * до конца, p2 будет указывать на нужный элемент. */
13    while (p1 != null) {
14      p1 = p1.next;
15      p2 = p2.next;
16      }
17      return p2;
18  }
```

Этот алгоритм выполняется за время $O(n)$ с затратами памяти $O(1)$.

2.3. Реализуйте алгоритм, удаляющий узел из середины односвязного списка (то есть узла, не являющегося ни начальным, ни конечным — не обязательно находящимся точно в середине). Доступ предоставляется только к этому узлу.

Пример:

Ввод: узел c из списка `a->b->c->d->e->f`

Вывод: ничего не возвращается, но новый список имеет вид: `a->b->d->e->f`

РЕШЕНИЕ

В этой задаче начало списка недоступно, а доступ предоставляется только к конкретному узлу. Задача решается простым копированием данных из следующего узла в текущий, с последующим удалением следующего узла.

Следующий код реализует этот алгоритм:

```
1  boolean deleteNode(LinkedListNode n) {
2    if (n == null || n.next == null) {
3      return false; // Ошибка
4    }
5    LinkedListNode next = n.next;
6    n.data = next.data;
7    n.next = next.next;
8    return true;
9  }
```

Если удаляемый узел является последним в связном списке, задача неразрешима. Ничего страшного — просто интервьюер хочет, чтобы вы обсудили с ним, что делать в этом случае. Можно, например, пометить данный узел как недопустимый.

2.4. Напишите код для разбиения связного списка вокруг значения x, так чтобы все узлы, меньшие x, предшествовали узлам, большим или равным x. Если x содержится в списке, то значения x должны следовать строго после элементов, меньших x (см. далее). Элемент разбивки x может находиться где угодно в «правой части»; он не обязан располагаться между левой и правой частью.

Пример:

Ввод: `3->5->8->5->10->2->1` [значение разбивки = 5]

Вывод: `3->1->2->10->5->5->8`

РЕШЕНИЕ

Если бы мы работали с массивом, то пришлось бы учитывать многочисленные сложности, связанные со смещением элементов. Операция сдвига в массиве очень затратна.

Со связным списком задача намного проще. Вместо того чтобы сдвигать и менять местами элементы, мы можем создать два разных связных списка: один для элементов, меньших x, а второй — для элементов, которые больше или равны x.

Мы проходим по списку, вставляя элементы в один из списков `before` или `after`. С достижением конца исходного связного списка и завершением разбивки можно выполнить слияние получившихся списков.

Этот метод в основном «стабилен» в том отношении, что элементы сохраняют исходный порядок, если не считать необходимых перемещений вокруг точки разбивки. Ниже приведена реализация этого алгоритма.

```
1  /* Передается начало связного списка и значение x для разбивки */
2  LinkedListNode partition(LinkedListNode node, int x) {
3    LinkedListNode beforeStart = null;
4    LinkedListNode beforeEnd = null;
5    LinkedListNode afterStart = null;
6    LinkedListNode afterEnd = null;
7
8    /* Разбивка списка */
9    while (node != null) {
10     LinkedListNode next = node.next;
```

```
11      node.next = null;
12      if (node.data < x) {
13        /* Узел вставляется в конец списка before */
14        if (beforeStart == null) {
15          beforeStart = node;
16          beforeEnd = beforeStart;
17        } else {
18          beforeEnd.next = node;
19          beforeEnd = node;
20      }
21      } else {
22        /* Узел вставляется в конец списка after */
23        if (afterStart == null) {
24          afterStart = node;
25          afterEnd = afterStart;
26        } else {
27          afterEnd.next = node;
28          afterEnd = node;
29      }
30     }
31      node = next;
32    }
33
34    if (beforeStart == null) {
35      return afterStart;
36    }
37
38    /* Слияние списков before и after */
39    beforeEnd.next = afterStart;
40    return beforeStart;
41  }
```

Если вам не нравится идея использования четырех переменных для отслеживания двух связных списков, вы не одиноки. Этот код можно немного укоротить.

Если вы не стремитесь сохранить «стабильность» элементов списка (строго говоря, это не обязательно, если интервьюер этого не требовал), вы можете изменить порядок элементов, наращивая список от начала и от конца.

При таком подходе создается «новый» список (с использованием существующих узлов). Элементы, большие элемента разбивки, помещаются в конец, а меньшие элементы помещаются в начало. Каждый раз, когда мы вставляем элемент, обновляется начало или конец списка.

```
1  LinkedListNode partition(LinkedListNode node, int x) {
2    LinkedListNode head = node;
3    LinkedListNode tail = node;
4
5    while (node != null) {
6      LinkedListNode next = node.next;
7      if (node.data < x) {
8        /* Вставить узел в начало. */
9        node.next = head;
10       head = node;
11     } else {
12       /* Вставить узел в конец. */
13       tail.next = node;
```

```
14          tail = node;
15       }
16        node = next;
17     }
18     tail.next = null;
19
20     // Начало изменилось, возвращаем его пользователю.
21     return head;
22  }
```

У этой задачи есть много равноэффективных решений. Если вы предложите что-то другое — это нормально!

2.5. Два числа хранятся в виде связных списков, в которых каждый узел представляет один разряд. Все цифры хранятся в *обратном* порядке, при этом младший разряд (единицы) хранится в начале списка. Напишите функцию, которая суммирует два числа и возвращает результат в виде связного списка.

Пример:

Ввод: `(7->1->6) + (5->9->2)`, то есть `617 + 295`.

Вывод: `2->1->9`, то есть `912`.

Дополнительно

Решите задачу, предполагая, что цифры записаны в прямом порядке.

Ввод: `(6->1->7) + (2->9->5)`, то есть `617 + 295`.

Вывод: `9->1->2`, то есть `912`.

РЕШЕНИЕ

В этой задаче полезно вспомнить, как именно работает сложение. Рассмотрим задачу:

```
  6 1 7
+ 2 9 5
```

Сначала мы суммируем 7 и 5, чтобы получить 12. Разряд 2 становится последним разрядом числа, а 1 переносится на следующий шаг. Затем мы суммируем 1, 1 и 9, чтобы получить 11. При этом 1 переходит в следующий разряд, а 1 остается на месте. Затем мы суммируем 1, 6 и 2, чтобы получить 9. Результат — 912.

Можно смоделировать этот процесс в рекурсивном виде, суммируя узел с узлом и перенося любые «избыточные» данные на следующий узел. Давайте рассмотрим этот метод на примере связного списка:

```
  7 -> 1 -> 6
+ 5 -> 9 -> 2
```

Нам нужно сделать следующее:

1. Сложить 7 и 5, получить 12. Двойка становится первым узлом нашего связного списка. Единица «запоминается» для следующей суммы:

```
List: 2 -> ?
```

2. Затем мы суммируем 1 и 9, прибавляем перенесенную единицу и получаем 11. Единица становится вторым элементом связного списка, а другая единица запоминается для следующей суммы:

```
List: 2 -> 1 -> ?
```

3. Наконец, суммируя 6, 2 и перенесенную единицу, мы получаем 9 — последний элемент нашего списка.

```
List: 2 -> 1 -> 9.
```

Следующий код реализует описанный алгоритм:

```
1  LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2, int carry) {
2    if (l1 == null && l2 == null && carry == 0) {
3      return null;
4    }
5
6    LinkedListNode result = new LinkedListNode();
7    int value = carry;
8    if (l1 != null) {
9      value += l1.data;
10   }
11   if (l2 != null) {
12     value += l2.data;
13   }
14
15   result.data = value % 10; /* Вторая цифра числа */
16
17   /* Рекурсия */
18   if (l1 != null || l2 != null) {
19     LinkedListNode more = addLists(l1 == null ? null : l1.next,
20                                    l2 == null ? null : l2.next,
21                                  value >= 10 ? 1 : 0);
22     result.setNext(more);
23   }
24   return result;
25 }
```

В этом коде необходимо принять меры предосторожности и обработать условие, когда один связный список короче другого, чтобы избежать исключений `null`-указателя!

Дополнительное условие

Концептуально алгоритм практически не изменяется (рекурсия, перенос), но его реализация несколько усложняется.

1. Один список может оказаться короче другого, а мы не можем обработать это «на лету». Например, представьте, что вы суммируете списки `(1 -> 2 -> 3 -> 4)` и `(5 -> 6 -> 7)`. Мы должны заранее знать, что элемент 5 должен сочетаться с 2, а не с 1. Для реализации этой проверки можно в самом начале сравнить длины списков и дополнить более короткий список нулями.
2. В основном решении задачи результаты добавлялись в конец списка. Это означает, что рекурсивный вызов должен *получить* перенос и возвратить результат (который будет присоединен в конец). Однако в данном случае результат добавляется в начало списка; рекурсивный вызов должен вернуть результат, как и прежде, а также перенос. Такое решение реализуется не так уж сложно, но оно становится более громоздким. Избавиться от этой проблемы поможет класс-обертка `PartialSum`.

Следующий код реализует наш алгоритм:

```
class PartialSum {
  public LinkedListNode sum = null;
  public int carry = 0;
}

LinkedListNode addLists(LinkedListNode l1, LinkedListNode l2) {
  int len1 = length(l1);
  int len2 = length(l2);

  /* Более короткий список дополняется нулями - см. примечание (1) */
  if (len1 < len2) {
    l1 = padList(l1, len2 - len1);
  } else {
    l2 = padList(l2, len1 - len2);
  }

  /* Суммирование списков */
  PartialSum sum = addListsHelper(l1, l2);

  /* Если осталось переносимое значение, вставить его в начало списка.
   * В противном случае просто вернуть связный список. */
  if (sum.carry == 0) {
    return sum.sum;
  } else {
    LinkedListNode result = insertBefore(sum.sum, sum.carry);
    return result;
  }
}

PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
  if (l1 == null && l2 == null) {
    PartialSum sum = new PartialSum();
    return sum;
  }
  /* Рекурсивное суммирование меньших цифр */
  PartialSum sum = addListsHelper(l1.next, l2.next);

  /* Добавление переноса к текущим данным */
  int val = sum.carry + l1.data + l2.data;

  /* Вставка текущей суммы */
  LinkedListNode full_result = insertBefore(sum.sum, val % 10);

  /* Вернуть текущую сумму и величину переноса */
  sum.sum = full_result;
  sum.carry = val / 10;
  return sum;
}

/* Дополнение списка нулями */
LinkedListNode padList(LinkedListNode l, int padding) {
  LinkedListNode head = l;
  for (int i = 0; i < padding; i++) {
    head = insertBefore(head, 0);
```

```
55    }
56    return head;
57  }
58
59  /* Вспомогательная функция для вставки узла в начало связного списка */
60  LinkedListNode insertBefore(LinkedListNode list, int data) {
61    LinkedListNode node = new LinkedListNode(data);
62    if (list != null) {
63      node.next = list;
64    }
65    return node;
66  }
```

Обратите внимание: код `insertBefore()`, `padList()` и `length` (не приводится) вынесен в отдельные методы. От этого программа становится более понятной и удобочитаемой, что немаловажно для собеседования.

2.6. Реализуйте функцию, проверяющую, является ли связный список палиндромом.

РЕШЕНИЕ

Для этой задачи можно представить палиндром в виде списка `0 -> 1 -> 2 -> 1 -> 0`. Мы знаем, что если список является палиндромом, то он должен одинаково читаться в прямом и обратном направлении. Так мы приходим к первому решению.

Решение 1. Перевернуть и сравнить

Первое решение — «перевернуть» связный список и сравнить обращенную версию с исходной. Если они одинаковы, значит, список является палиндромом.

Обратите внимание: достаточно сравнить первые половины списков. Если первая половина нормального списка совпадает с первой половиной обращенного, то и вторые половины совпадут.

```
1  class HeadAndTail {
2    public LinkedListNode head;
3    public LinkedListNode tail;
4    public HeadAndTail(LinkedListNode h, LinkedListNode t) {
5      head = h;
6      tail = t;
7    }
8  }
9
10 boolean isPalindrome(LinkedListNode head) {
11   HeadAndTail reversed = reverse(head);
12   LinkedListNode reversedHead = reversed.head;
13   return isEqual(head, reversedHead);
14 }
15
16 HeadAndTail reverse(LinkedListNode head) {
17   if (head == null) {
18     return null;
19   }
20   HeadAndTail ht = reverse(head.next);
```

```
21    LinkedListNode clonedHead = head.clone();
22    clonedHead.next = null;
23
24    if (ht == null) {
25      return new HeadAndTail(clonedHead, clonedHead);
26    }
27    ht.tail.next = clonedHead;
28    return new HeadAndTail(ht.head, clonedHead);
29  }
30
31  boolean isEqual(LinkedListNode one, LinkedListNode two) {
32    LinkedListNode head1 = one;
33    LinkedListNode head2 = two;
34    while (head1 != null && head2 != null) {
35      if (head1.data != head2.data) {
36        return false;
37     }
38      head1 = head1.next;
39      head2 = head2.next;
40    }
41    return head1 == null && head2 == null;
42  }
```

Обратите внимание: часть кода была выделена в функции `reverse` и `isEqual`. Также мы создали новый класс для возвращения начала и конца. С таким же успехом можно было вернуть массив из двух элементов, но такое решение создает больше сложностей с сопровождением.

Решение 2. Итеративный подход

Необходимо найти связные списки, в которых первая половина списка является результатом инверсии второй половины. Как это сделать? Путем инвертирования первой половины списка. Стек поможет нам в этом.

Первая половина элементов заносится в стек. Это можно сделать двумя разными способами в зависимости от того, известен ли нам размер связного списка.

Если размер связного списка известен, можно перебрать первую половину элементов в стандартном цикле `for`, помещая каждый элемент в стек. Конечно, при этом необходима осторожность: следует учитывать, что длина связного списка может быть нечетной.

Если размер связного списка неизвестен, для перебора можно воспользоваться методом быстрого/медленного указателя, описанным в начале главы. На каждой итерации цикла данные медленного указателя помещаются в стек. Когда быстрый указатель дойдет до конца списка, медленный окажется в его середине. В этот момент в стеке находятся все элементы первой половины связного списка, но расположенные в обратном порядке.

Теперь нужно просто перебрать оставшуюся часть списка. При каждой итерации узел сравнивается с вершиной стека. Если процесс закончился без обнаружения различий, значит, связный список является палиндромом.

```
1  boolean isPalindrome(LinkedListNode head) {
2    LinkedListNode fast = head;
3    LinkedListNode slow = head;
4
5    Stack<Integer> stack = new Stack<Integer>();
```

```
6
7     /* Элементы первой половины связного списка помещаются в стек. Когда
8      * быстрый указатель (со скоростью 2x) достигает конца связного списка,
9      * медленный находится в середине */
10    while (fast != null && fast.next != null) {
11      stack.push(slow.data);
12      slow = slow.next;
13      fast = fast.next.next;
14    }
15
16    /* Нечетное количество элементов, средний элемент пропускается */
17    if (fast != null) {
18      slow = slow.next;
19    }
20
21    while (slow != null) {
22      int top = stack.pop().intValue();
23
24      /* Если значения не совпадают, это не палиндром */
25      if (top != slow.data) {
26        return false;
27     }
28      slow = slow.next;
29    }
30    return true;
31  }
```

Решение 3. Рекурсивный метод

Пара слов по поводу обозначений: в этом решении будет использоваться запись Kx, где K — значение узла, а x (принимает значение `f` или `b`) указывает, к какому узлу мы обращаемся — из начала списка (`f`, `front`) или из конца (`b`, `back`). Например, в приведенном ниже связном списке запись `2b` обозначает второй (с конца) узел со значением 2.

Как и многие другие задачи по связным спискам, эта задача может быть решена рекурсивно. Нужно сравнить элементы 0 и `n-1`, 1 и `n-2`, 2 и `n-3` и так далее до центрального элемента. Например:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

Чтобы применить этот метод, нужно сначала узнать о достижении среднего элемента; тогда задача будет сведена к описанному базовому случаю. Для этого можно каждый раз передавать `length-2` в качестве длины. Если длина равна 0 или 1, мы находимся в центре связного списка (так как длина каждый раз уменьшается на 2). После `N`/2 рекурсий значение `length` уменьшится до 0.

```
1  recurse(Node n, int length) {
2    if (length == 0 || length == 1) {
3      return [something]; // At middle
4    }
5    recurse(n.next, length - 2);
6    ...
7  }
```

Этот метод лежит в основе метода `isPalindrome`. Суть алгоритма — сравнение узла `i` с узлом `n-i` (чтобы проверить, является ли список палиндромом). Как это сделать?

Стек вызовов выглядит примерно так:

```
1  v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
2    v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
3      v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
4        v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
5        возвращается v3
6      возвращается v2
7    возвращается v1
8  возвращается ?
```

В этом стеке каждый вызов проверяет, является ли список палиндромом, сравнивая его начальный узел с соответствующим узлом с конца списка:

- строка 1: сравнивает узел `0f` с узлом `0b`;
- строка 2: сравнивает узел `1f` с узлом `1b`;
- строка 3: сравнивает узел `2f` с узлом `2b`;
- строка 4: сравнивает узел `3f` с узлом `3b`.

Выполним раскрутку стека с возвращением узлов так, как описано ниже:

- строка 4 «видит», что узел является средним (начиная с длины 1), и возвращает `head.next`. Значение `head` соответствует узлу 3, поэтому `head.next` является узел `2b`;
- строка 3 сравнивает узел `2f` с `returned_node` (значение, полученное от предыдущего рекурсивного вызова), который является узлом `2b`. Если значения совпадают, ссылка на узел `1b` (`returned_node.next`) передается строке 2;
- строка 2 сравнивает узел `1f` с `returned_node` (узел `1b`). Если значения совпадают, ссылка на `0b` (или `returned_node.next`) передается строке 1;
- строка 1 сравнивает узел `0f` с `returned_node` (узел `0b`). Если значения совпадают, возвращается `true`.

В общем виде: каждый вызов сравнивает свое начало с `returned_node` и затем передает `returned_node.next` далее по стеку. Каждый узел `i` сравнивается с узлом `n-i`. Если в какой-либо точке они не совпадают, возвращается `false`; это значение проверяется всеми вызовами далее по стеку.

Но постойте, ведь иногда мы говорим, что возвращается значение `boolean`, а иногда — `node`? Так что из двух?

И то и другое. Мы создаем простой класс с двумя полями, `boolean` и `node`, и возвращаем экземпляр этого класса.

```
1  class Result {
2    public LinkedListNode node;
3    public boolean result;
4  }
```

Следующий пример демонстрирует параметры и возвращаемые значения для приведенного выше списка:

```
1  isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
2    isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
3      isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
```

```
4        isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
5       isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
6          returns node 3b, true
7        returns node 2b, true
8      returns node 1b, true
9    returns node 0b, true
10  returns null, true
```

Теперь остается лишь доработать технические подробности:

```
1  boolean isPalindrome(LinkedListNode head) {
2    int length = lengthOfList(head);
3    Result p = isPalindromeRecurse(head, length);
4    return p.result;
5  }
6
7  Result isPalindromeRecurse(LinkedListNode head, int length) {
8    if (head == null || length <= 0) { // Четное количество узлов
9      return new Result(head, true);
10   } else if (length == 1) { // Нечетное количество узлов
11     return new Result(head.next, true);
12   }
13
14   /* Рекурсия для подсписка. */
15   Result res = isPalindromeRecurse(head.next, length - 2);
16
17   /* Если рекурсивные вызовы показывают, что это не палиндром,
18    * вернуть признак неудачной проверки. */
19   if (!res.result || res.node == null) {
20     return res;
21   }
22
23   /* Сравнить с соответствующим узлом с другой стороны. */
24   res.result = (head.data == res.node.data);
25
26   /* Вернуть следующий узел. */
27   res.node = res.node.next;
28
29   return res;
30 }
31
32 int lengthOfList(LinkedListNode n) {
33   int size = 0;
34   while (n != null) {
35     size++;
36     n = n.next;
37   }
38   return size;
39 }
```

Почему нам пришлось пойти на хлопоты с созданием специального класса `Result`? Нет ли лучшего способа? Нет. По крайней мере, в Java.

Если вы пишете код на C или C++, можно было передать двойной указатель:

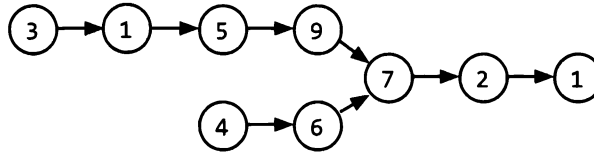
```
1  bool isPalindromeRecurse(Node head, int length, Node** next) {
2    ...
3  }
```

Выглядит некрасиво, но работает.

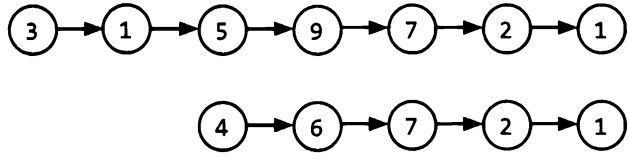
2.7. Проверьте, пересекаются ли два заданных (одно-)связных списка. Верните узел пересечения. Учтите, что пересечение определяется ссылкой, а не значением. Иначе говоря, если *k*-й узел первого связного списка точно совпадает (по ссылке) с *j*-м узлом второго связного списка, то списки считаются пересекающимися.

РЕШЕНИЕ

Диаграмма с пересекающимися связными списками поможет лучше понять суть происходящего:



А на этой диаграмме изображены непересекающиеся связные списки:



Учтите, что связные списки не обязаны иметь одинаковую длину, это всего лишь частный случай.

Для начала разберемся, как определить, пересекаются ли два связных списка.

Проверка существования пересечения

Как определить, пересекаются ли два списка? Одно из возможных решений — воспользоваться хеш-таблицей и занести в нее все узлы списков. Будьте внимательны: в качестве ключа должно использоваться местонахождение узла в памяти, а не его значение.

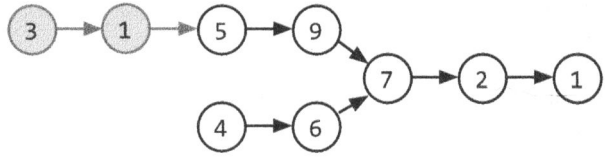
Впрочем, существует и более простой способ. Обратите внимание на то, что у двух пересекающихся связных списков последний узел всегда совпадает. Следовательно, мы можем просто добраться до конца каждого связного списка и сравнить последние узлы.

Но как определить, где находится пересечение?

Поиск узла пересечения

Инстинктивно хочется пройти по каждому связному списку в обратном направлении. Точка, в которой связные списки «разойдутся», и будет пересечением. Конечно, в односвязном списке перемещение в обратном направлении невозможно.

Если бы связные списки имели одинаковую длину, можно было бы просто синхронно перемещаться по ним. Обнаружив точку схождения, вы найдете пересечение.



Если длины списков различаются, по сути, нужно «отсечь», или просто проигнорировать, лишние (серые) узлы.

Как это сделать? Если длины двух списков известны, то количество отсекаемых узлов равно разности между ними.

Длины можно получить одновременно с нахождением конечных узлов связных списков (которые использовались на первом шаге для поиска пересечения).

Все вместе

В результате мы приходим к алгоритму, состоящему из нескольких шагов.

1. Пройти по элементам каждого связного списка для определения длин и нахождения конечных узлов.
2. Сравнить конечные узлы. Если они различны (по ссылке, не по значению), немедленно вернуть управление — пересечение отсутствует.
3. Установить два указателя в начало каждого из связных списков.
4. В более длинном связном списке переместить указатель на разность длин.
5. Перебирать связный список до тех пор, пока указатели не совпадут.

Ниже приведена реализация этого алгоритма.

```
1  LinkedListNode findIntersection(LinkedListNode list1, LinkedListNode list2) {
2    if (list1 == null || list2 == null) return null;
3
4    /* Получить конечные узлы и размеры. */
5    Result result1 = getTailAndSize(list1);
6    Result result2 = getTailAndSize(list2);
7
8    /* Если конечные узлы различны, пересечения нет. */
9    if (result1.tail != result2.tail) {
10     return null;
11   }
12
13   /* Указатели устанавливаются в начало каждого связного списка. */
14   LinkedListNode shorter = result1.size < result2.size ? list1 : list2;
15   LinkedListNode longer = result1.size < result2.size ? list2 : list1;
16
17   /* Указатель смещается по длинному списку на разность длин. */
18   longer = getKthNode(longer, Math.abs(result1.size - result2.size));
19
20   /* Указатели перемещаются до схождения. */
21   while (shorter != longer) {
22     shorter = shorter.next;
23     longer = longer.next;
24   }
25
26   /* Возвращается любой из списков. */
```

```
27    return longer;
28  }
29
30  class Result {
31    public LinkedListNode tail;
32    public int size;
33    public Result(LinkedListNode tail, int size) {
34      this.tail = tail;
35      this.size = size;
36    }
37  }
38
39  Result getTailAndSize(LinkedListNode list) {
40    if (list == null) return null;
41
42    int size = 1;
43    LinkedListNode current = list;
44    while (current.next != null) {
45      size++;
46      current = current.next;
47    }
48    return new Result(current, size);
49  }
50
51  LinkedListNode getKthNode(LinkedListNode head, int k) {
52    LinkedListNode current = head;
53    while (k > 0 && current != null) {
54      current = current.next;
55      k--;
56    }
57    return current;
58  }
```

Алгоритм выполняется за время $O(A+B)$, где A и B — длины двух связных списков, и увеличивает затраты памяти на $O(1)$.

2.8. Для кольцевого связного списка реализуйте алгоритм, возвращающий начальный узел петли.

Определение

Кольцевой связный список — это связный список, в котором указатель следующего узла ссылается на более ранний узел, образуя петлю.

Пример:

Ввод: `A->B->C->D->E->C` (предыдущий узел `C`)

Вывод: `C`

РЕШЕНИЕ

Эта задача является разновидностью классической задачи, задаваемой на собеседованиях, — определить, содержит ли связный список петлю.

Часть 1. Определяем, есть ли в связном списке петля

Простейший способ выяснить, есть ли в связном списке петля, — использовать метод быстрого/медленного указателя. `FastRunner` делает два шага за один такт,

а SlowRunner — только один. Подобно двум гоночным автомобилям, мчащимся по одной трассе разными путями, через какое-то время они непременно должны встретиться.

Проницательный читатель может задать вопрос: может ли быстрый указатель «перепрыгнуть» медленный без столкновения? Это невозможно. Допустим, что FastRunner *перепрыгнул* через SlowRunner и теперь находится в элементе i+1 (а медленный — в i). Это означает, что на предыдущем шаге указатель SlowRunner был в точке i-1, а FastRunner — ((i+1)-2)=i-1. Следовательно, столкновение неизбежно.

Часть 2. Когда же они встретятся?

Допустим, что связный список содержит часть размера k, не замкнутую в петлю.

Как узнать, когда встретятся FastRunner и SlowRunner, используя алгоритм из части 1?

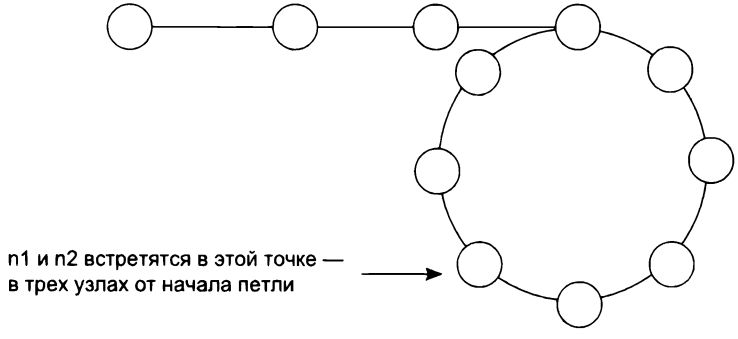
Мы знаем, что за каждые p шагов, на которые перемещается SlowRunner, указатель FastRunner делает 2p шагов. Следовательно, когда SlowRunner через k шагов попадет в петлю, FastRunner пройдет 2k шагов, а следовательно, войдет в петлю на расстояние k. Поскольку k может быть существенно больше, чем длина петли, введем обозначение K=mod(k, РАЗМЕР_ПЕТЛИ).

В каждом последующем шаге FastRunner и SlowRunner расходятся на шаг (или сближаются на шаг — зависит от точки зрения). Поскольку движение идет по кругу, когда указатель A отходит от B на q шагов, он также становится на q шагов ближе к B.

Итак, мы знаем следующие факты:

1. SlowRunner: 0 шагов внутри петли.
2. FastRunner: K шагов внутри петли.
3. SlowRunner: отстает от FastRunner на K шагов.
4. FastRunner: отстает от SlowRunner на РАЗМЕР_ПЕТЛИ-K шагов.
5. FastRunner нагоняет SlowRunner со скоростью 1 шаг за единицу времени.

Когда же они встретятся? Если FastRunner на РАЗМЕР_ПЕТЛИ-K шагов отстает от SlowRunner, а FastRunner нагоняет его со скоростью 1 шаг за единицу времени, они встретятся через РАЗМЕР_ПЕТЛИ-K шагов. В этой точке они будут отстоять на K шагов от начала петли. Назовем эту точку CollisionSpot.



Часть 3. Как найти начало петли?

Мы теперь знаем, что CollisionSpot находится за K узлов до начала петли. Поскольку K=mod(k,РАЗМЕР_ПЕТЛИ) (или k=K+M*РАЗМЕР_ПЕТЛИ для любого целого M), можно сказать, что до начала петли k узлов. Например, если узел N смещен на 2 узла в петле из 5 элементов, то можно сказать, что он смещен на 7, 12 и даже 397 узлов в петле.

Следовательно, и CollisionSpot, и LinkedListHead находятся в k узлах от начала петли.

Если мы сохраним один указатель в CollisionSpot и переместим другой в LinkedListHead, то каждый из них будет отстоять на k узлов от LoopStart. Одновременное перемещение этих указателей приведет к их столкновению — на сей раз через k шагов — в точке LoopStart. Остается лишь вернуть этот узел.

Часть 4. Собираем все воедино

FastPointer двигается в два раза быстрее, чем SlowPointer. Через k узлов SlowPointer оказывается в петле, а FastPointer — на k-м узле связного списка. Это означает, что FastPointer и SlowPointer отделяют друг от друга РАЗМЕР_ПЕТЛИ-k узлов.

Если FastPointer двигается на 2 узла за одиночный шаг SlowPointer, указатели будут сближаться на каждом цикле и встретятся через РАЗМЕР_ПЕТЛИ-k тактов. В этой точке они окажутся на расстоянии k узлов от начала петли.

Начало связного списка расположено в k узлах от начала петли. Следовательно, если мы сохраним быстрый указатель в текущей позиции, а затем переместим медленный в начало связного списка, точка встречи окажется в начале петли.

Давайте запишем алгоритм, воспользовавшись информацией из частей 1–3:

1. Создадим два указателя: FastPointer и SlowPointer.
2. Будем перемещать FastPointer на 2 шага, а SlowPointer на 1 шаг.
3. Когда указатели встретятся, нужно переместить SlowPointer в LinkedListHead, а FastPointer оставить на том же месте.
4. SlowPointer и FastPointer продолжают двигаться со своими скоростями, точка их следующей встречи будет искомым результатом.

Следующий код реализует описанный алгоритм:

```
1  LinkedListNode FindBeginning(LinkedListNode head) {
2    LinkedListNode slow = head;
3    LinkedListNode fast = head;
4
5    /* Поиск точки встречи (РАЗМЕР_ПЕТЛИ - k шагов в связном списке) */
6    while (fast != null && fast.next != null) {
7      slow = slow.next;
8      fast = fast.next.next;
9      if (slow == fast) { // Встреча
10       break;
11     }
12   }
13
14   /* Проверка ошибок - точки встречи нет, а значит, нет и петли */
15   if (fast == null || fast.next == null) {
```

```
16      return null;
17    }
18
19    /* Slow перемещается в начало, fast остается в точке встречи. Если
20     * они будут двигаться в одном темпе, то встретятся в начале петли. */
21    slow = head;
22    while (slow != fast) {
23      slow = slow.next;
24      fast = fast.next;
25    }
26
27    /* Оба указателя находятся в начале петли. */
28    return fast;
29  }
```

3

Стеки и очереди

3.1. Опишите, как можно использовать один одномерный массив для реализации трех стеков.

РЕШЕНИЕ

Подобно многим задачам, все зависит от того, как мы собираемся организовать эти стеки. Если достаточно выделить фиксированное пространство для каждого стека, можно так и поступить. Но в этом случае один из стеков может оказаться практически исчерпанным, тогда как другие будут практически пустыми.

Также можно использовать более гибкую систему разделения пространства, но это значительно повысит сложность задачи.

Подход 1. Фиксированное разделение

Можно разделить массив на три равные части и разрешить стекам расширяться в пределах ограниченного пространства. Далее границы диапазонов будут описываться скобками: квадратные скобки `[]` означают, что граничные значения входят в диапазон, а с круглыми скобками `()` граничные значения в диапазон не входят.

- Стек 1: `[0, n/3)`.
- Стек 2: `[n/3, 2n/3)`.
- Стек 3: `[2n/3, n)`.

Код этого решения приведен ниже:

```
1  class FixedMultiStack {
2    private int numberOfStacks = 3;
3    private int stackCapacity;
4    private int[] values;
5    private int[] sizes;
6
7    public FixedMultiStack(int stackSize) {
8      stackCapacity = stackSize;
9      values = new int[stackSize * numberOfStacks];
10      sizes = new int[numberOfStacks];
11    }
12
13    /* Занесение значения в стек. */
14    public void push(int stackNum, int value) throws FullStackException {
15      /* Проверка наличия места для следующего элемента */
16      if (isFull(stackNum)) {
17        throw new FullStackException();
18     }
```

```
19
20      /* Увеличение указателя стека с последующим обновлением вершины. */
21      sizes[stackNum]++;
22      values[indexOfTop(stackNum)] = value;
23    }
24
25    /* Извлечение элемента с вершины стека. */
26    public int pop(int stackNum) {
27      if (isEmpty(stackNum)) {
28        throw new EmptyStackException();
29     }
30
31      int topIndex = indexOfTop(stackNum);
32      int value = values[topIndex]; // Получение вершины
33      values[topIndex] = 0; // Очистка
34      sizes[stackNum]--; // Сокращение
35      return value;
36    }
37
38    /* Return top element. */
39    public int peek(int stackNum) {
40      if (isEmpty(stackNum)) {
41        throw new EmptyStackException();
42      }
43      return values[indexOfTop(stackNum)];
44    }
45
46    /* Проверка пустого стека. */
47    public boolean isEmpty(int stackNum) {
48      return sizes[stackNum] == 0;
49    }
50
51    /* Проверка заполненного стека. */
52    public boolean isFull(int stackNum) {
53      return sizes[stackNum] == stackCapacity;
54    }
55
56    /* Получение индекса вершины стека. */
57    private int indexOfTop(int stackNum) {
58      int offset = stackNum * stackCapacity;
59      int size = sizes[stackNum];
60      return offset + size - 1;
61    }
62  }
```

При наличии дополнительной информации о предполагаемом использовании стеков алгоритм можно изменить. Например, если предполагается, что в стеке 1 будет больше элементов, чем в стеке 2, можно перераспределить пространство в пользу стека 1.

Подход 2. Гибкое разделение

Второй подход — динамическое выделение пространства для блоков стека. Когда один из стеков перестает помещаться в исходном пространстве, его емкость увеличивается, а элементы сдвигаются по мере необходимости.

Кроме того, можно создать массив таким образом, чтобы последний стек начинался в конце массива и заканчивался в начале, то есть сделать его циклическим.

Код решения получается намного более сложным, чем уместно на собеседовании. Скорее всего, вы ограничитесь написанием псевдокода или кода отдельных компонентов.

```
1  public class MultiStack {
2    /* StackInfo - простой класс для хранения информации о каждом стеке.
3     * То же самое можно сделать при помощи отдельных переменных, но такое
4     *  решение получается громоздким и ничего реально не дает. */
6    private class StackInfo {
7      public int start, size, capacity;
8      public StackInfo(int start, int capacity) {
9        this.start = start;
10       this.capacity = capacity;
11     }
12
13     /* Проверить, лежит ли индекс в границах стека.
14      * Стек может продолжаться от начала массива. */
15     public boolean isWithinStackCapacity(int index) {
16       /* Если индекс выходит за границы массива, вернуть false. */
17       if (index < 0 || index >= values.length) {
18         return false;
19     }
20
21       /* При циклическом переносе индекса внести поправку. */
22       int contiguousIndex = index < start ? index + values.length : index;
23       int end = start + capacity;
24       return start <= contiguousIndex && contiguousIndex < end;
25     }
26
27     public int lastCapacityIndex() {
28       return adjustIndex(start + capacity - 1);
29     }
30
31     public int lastElementIndex() {
32       return adjustIndex(start + size - 1);
33     }
34
35     public boolean isFull() { return size == capacity; }
36     public boolean isEmpty() { return size == 0; }
37   }
38
39   private StackInfo[] info;
40   private int[] values;
41
42   public MultiStack(int numberOfStacks, int defaultSize) {
43     /* Создание метаданных для всех стеков. */
44     info = new StackInfo[numberOfStacks];
45     for (int i = 0; i < numberOfStacks; i++) {
46       info[i] = new StackInfo(defaultSize * i, defaultSize);
47     }
48     values = new int[numberOfStacks * defaultSize];
49   }
50
51   /* Занесение значения в стек со сдвигом/расширением стеков по мере
```

```
52     * необходимости. Если все стеки заполнены, выдается исключение. */
53    public void push(int stackNum, int value) throws FullStackException {
54      if (allStacksAreFull()) {
55        throw new FullStackException();
56     }
57
58      /* Если стек заполнен, расширить его. */
59      StackInfo stack = info[stackNum];
60      if (stack.isFull()) {
61        expand(stackNum);
62     }
63
64      /* Определение индекса верхнего элемента в массиве + 1
65       * и увеличение указателя стека */
66      stack.size++;
67      values[stack.lastElementIndex()] = value;
68    }
69
70    /* Извлечение из стека. */
71    public int pop(int stackNum) throws Exception {
72      StackInfo stack = info[stackNum];
73      if (stack.isEmpty()) {
74        throw new EmptyStackException();
75     }
76
77      /* Удаление последнего элемента. */
78      int value = values[stack.lastElementIndex()];
79      values[stack.lastElementIndex()] = 0; // Очистка
80      stack.size--; // Уменьшение размера
81      return value;
82    }
83
84    /* Получение верхнего элемента стека.*/
85    public int peek(int stackNum) {
86      StackInfo stack = info[stackNum];
87      return values[stack.lastElementIndex()];
88    }
89    /* Сдвиг элементов в стеке на один элемент. При наличии свободного места
90     * стек уменьшится на один элемент. Если свободного места нет,
91     * также нужно будет выполнить сдвиг в следующем стеке. */
92    private void shift(int stackNum) {
93      System.out.println("/// Shifting " + stackNum);
94      StackInfo stack = info[stackNum];
95
96      /* Если стек заполнен, следующий стек необходимо сдвинуть на один
97       * элемент. Текущий стек занимает освободившийся индекс. */
98      if (stack.size >= stack.capacity) {
99        int nextStack = (stackNum + 1) % info.length;
100       shift(nextStack);
101       stack.capacity++; // Захват индекса из следующего стека
102     }
103
104     /* Сдвиг всех элементов в стеке. */
105     int index = stack.lastCapacityIndex();
106     while (stack.isWithinStackCapacity(index)) {
107       values[index] = values[previousIndex(index)];
108       index = previousIndex(index);
```

```
109     }
110
111     /* Изменение данных стека. */
112     values[stack.start] = 0; // Очистка
113     stack.start = nextIndex(stack.start); // Перемещение start
114     stack.capacity--; // Уменьшение емкости
115   }
116
117   /* Расширение стека посредством сдвига других стеков. */
118   private void expand(int stackNum) {
119     shift((stackNum + 1) % info.length);
120     info[stackNum].capacity++;
121   }
122
123   /* Возвращается фактическое количество элементов в стеке. */
124   public int numberOfElements() {
125     int size = 0;
126     for (StackInfo sd : info) {
127       size += sd.size;
128     }
129     return size;
130   }
131
132   /* Возвращает true, если все стеки заполнены. */
133   public boolean allStacksAreFull() {
134     return numberOfElements() == values.length;
135   }
136
137   /* Индекс переводится в диапазон 0 -> length - 1. */
138   private int adjustIndex(int index) {
139     /* Оператор Java mod может возвращать отрицательные значения.
140      * Например, (-11 % 5) вернет -1, а не 4, как требуется
141      * (из-за переноса индекса). */
142     int max = values.length;
143     return ((index % max) + max) % max;
144   }
145
146   /* Получение следующего индекса с поправкой на перенос. */
147   private int nextIndex(int index) {
148     return adjustIndex(index + 1);
149   }
150
151   /* Получение предыдущего индекса с поправкой на перенос. */
152   private int previousIndex(int index) {
153     return adjustIndex(index - 1);
154   }
155 }
```

В подобных задачах важно сосредоточиться на написании чистого, простого в сопровождении кода. Используйте дополнительные классы, как мы сделали со `StackData`, и выделяйте блоки кода в отдельные методы. Этот совет пригодится не только для прохождения собеседования, но и в реальной работе.

3.2. Как реализовать стек, в котором кроме стандартных функций `push` и `pop` будет поддерживаться функция `min`, возвращающая минимальный элемент? Все операции — `push`, `pop` и `min` — должны выполняться за время $O(1)$.

РЕШЕНИЕ

Максимумы и минимумы изменяются относительно редко, а точнее, они могут измениться только при добавлении нового элемента.

Одно из возможных решений — завести одно значение `minValue` типа `int`, которое добавляется в качестве поля `Stack`. При извлечении `minValue` из стека осуществляется поиск нового минимума по стеку. К сожалению, такая реализация нарушает ограничение на время выполнения — $O(1)$.

Чтобы лучше понять проблему, рассмотрим небольшой пример:

```
push(5); // стек: {5}, минимум: 5
push(6); // стек: {6, 5}, минимум: 5
push(3); // стек: {3, 6, 5}, минимум: 3
push(7); // стек: {7, 3, 6, 5}, минимум: 3
pop(); // извлекается 7. Стек: {3, 6, 5}, минимум: 3
pop(); // извлекается 3. Стек: {6, 5}. минимум: 5.
```

Обратите внимание: когда стек возвращается в предыдущее состояние (`{6,5}`), минимум также должен вернуться в предшествующее состояние (`5`). Этот факт подсказывает второй вариант решения этой задачи.

Отслеживая минимум в каждом состоянии, можно легко узнать минимальный элемент. Можно, например, записывать для каждого узла текущий минимальный элемент. Затем, чтобы найти `min`, достаточно посмотреть, какое значение является минимумом для верхнего элемента.

При занесении элемента в стек вычисляется текущий минимум, который присваивается в качестве «локального минимума» этого элемента.

```
1  public class StackWithMin extends Stack<NodeWithMin> {
2    public void push(int value) {
3      int newMin = Math.min(value, min());
4      super.push(new NodeWithMin(value, newMin));
5    }
6
7    public int min() {
8      if (this.isEmpty()) {
9        return Integer.MAX_VALUE; // Код ошибки
10     } else {
11       return peek().min;
12     }
13   }
14 }
15
16 class NodeWithMin {
17   public int value;
18   public int min;
19   public NodeWithMin(int v, int min){
20     value = v;
21     this.min = min;
22   }
23 }
```

У решения есть один недостаток: при очень большом стеке хранение минимума для каждого элемента приведет к значительным затратам памяти. Существует ли лучшее решение?

Эффективность можно немного повысить за счет использования дополнительного стека, в котором хранятся минимумы.

```
1  public class StackWithMin2 extends Stack<Integer> {
2    Stack<Integer> s2;
3    public StackWithMin2() {
4      s2 = new Stack<Integer>();
5    }
6
7    public void push(int value){
8      if (value <= min()) {
9        s2.push(value);
10     }
11     super.push(value);
12   }
13
14   public Integer pop() {
15     int value = super.pop();
16     if (value == min()) {
17       s2.pop();
18     }
19     return value;
20   }
21
22   public int min() {
23     if (s2.isEmpty()) {
24       return Integer.MAX_VALUE;
25     } else {
26       return s2.peek();
27     }
28   }
29 }
```

Почему такое решение более эффективно по затратам памяти? Допустим, вы работаете с огромным стеком, в котором первый вставленный элемент оказывается минимумом. В первом решении придется хранить `n` целых чисел, где `n` — размер стека. Во втором решении достаточно сохранить несколько фрагментов данных: второй стек с элементом и поля в этом стеке.

3.3. Как известно, слишком высокая стопка тарелок может развалиться. Следовательно, в реальной жизни, когда высота стопки превысила бы некоторое пороговое значение, мы начали бы складывать тарелки в новую стопку. Реализуйте структуру данных `SetOfStacks`, имитирующую реальную ситуацию. Структура `SetOfStack` должна состоять из нескольких стеков, новый стек создается, как только предыдущий достигнет порогового значения. Методы `SetOfStacks.push()` и `SetOfStacks.pop()` должны вести себя так же, как при работе с одним стеком (то есть метод `pop()` должен возвращать те же значения, которые бы он возвращал при использовании одного большого стека).

Дополнительно

Реализуйте функцию `popAt(int index)`, которая осуществляет операцию pop с заданным внутренним стеком.

РЕШЕНИЕ

В соответствии с формулировкой задачи структура данных будет иметь следующий вид:

```
1  class SetOfStacks {
2    ArrayList<Stack> stacks = new ArrayList<Stack>();
3    public void push(int v) { ... }
4    public int pop() { ... }
5  }
```

Известно, что `push()` должен работать так же, как для одиночного стека; это означает, что его реализация должна вызывать `push()` для последнего стека из массива стеков. При этом нужно действовать аккуратно: если последний стек заполнен, нужно создать новый стек. Код будет выглядеть примерно так:

```
1  void push(int v) {
2    Stack last = getLastStack();
3    if (last != null && !last.isFull()) { // Добавить в последний стек
4      last.push(v);
5    } else { // Необходимо создать новый стек
6      Stack stack = new Stack(capacity);
7      stack.push(v);
8      stacks.add(stack);
9    }
10 }
```

Что должен делать метод `pop()`? Он, как и `push()`, должен работать с последним стеком. Если последний стек после удаления элемента методом `pop()` оказывается пустым, его нужно удалить из списка стеков:

```
1  int pop() {
2    Stack last = getLastStack();
3    if (last == null) throw new EmptyStackException();
4    int v = last.pop();
5    if (last.size == 0) stacks.remove(stacks.size() - 1);
6    return v;
7  }
```

Дополнительно: реализация popAt(int index)

Извлечение из внутреннего стека реализуется несколько сложнее. Представьте себе систему с «переносом»: если требуется извлечь элемент из стека 1, то из стека 2 нужно удалить *нижний элемент* и занести его в стек 1. Затем аналогичная операция должна быть проделана со стеками 3 и 2, 4 и 3 и т. д.

На это можно возразить, что вместо «переноса» можно смириться с тем, что некоторые стеки заполнены не на всю емкость. Этот компромисс улучшит временную сложность (притом заметно при большом количестве элементов), но может создать проблемы потом, если пользователь предположит, что все стеки (кроме последнего) работают на полной емкости. Однозначного правильного решения в такой ситуации нет; обсудите этот вопрос с интервьюером.

```
1  public class SetOfStacks {
2    ArrayList<Stack> stacks = new ArrayList<Stack>();
3    public int capacity;
```

```
4    public SetOfStacks(int capacity) {
5      this.capacity = capacity;
6    }
7
8    public Stack getLastStack() {
9      if (stacks.size() == 0) return null;
10     return stacks.get(stacks.size() - 1);
11   }
12
13   public void push(int v) { /* см. выше */ }
14   public int pop() { /* см. выше */ }
15   public boolean isEmpty() {
16     Stack last = getLastStack();
17     return last == null || last.isEmpty();
18   }
19
20   public int popAt(int index) {
21     return leftShift(index, true);
22   }
23
24   public int leftShift(int index, boolean removeTop) {
25     Stack stack = stacks.get(index);
26     int removed_item;
27     if (removeTop) removed_item = stack.pop();
28     else removed_item = stack.removeBottom();
29     if (stack.isEmpty()) {
30       stacks.remove(index);
31     } else if (stacks.size() > index + 1) {
32       int v = leftShift(index + 1, false);
33       stack.push(v);
34     }
35     return removed_item;
36   }
37 }
38
39 public class Stack {
40   private int capacity;
41   public Node top, bottom;
42   public int size = 0;
43
44   public Stack(int capacity) { this.capacity = capacity; }
45   public boolean isFull() { return capacity == size; }
46
47   public void join(Node above, Node below) {
48     if (below != null) below.above = above;
49     if (above != null) above.below = below;
50   }
51
52   public boolean push(int v) {
53     if (size >= capacity) return false;
54     size++;
55     Node n = new Node(v);
56     if (size == 1) bottom = n;
57     join(n, top);
58     top = n;
59     return true;
60   }
61
```

```
62    public int pop() {
63      Node t = top;
64      top = top.below;
65      size--;
66      return t.value;
67    }
68
69    public boolean isEmpty() {
70      return size == 0;
71    }
72
73    public int removeBottom() {
74      Node b = bottom;
75      bottom = bottom.above;
76      if (bottom != null) bottom.below = null;
77      size--;
78      return b.value;
79    }
80  }
```

Концептуально задача не сложна, но ее полная реализация потребует написания довольно объемного кода. Вряд ли интервьюер захочет, чтобы вы написали весь код.

Хорошая стратегия в подобных задачах — разделение кода на методы. Например, в нашем коде присутствует метод `leftShift`, который вызывается в методе `popAt`. Этот прием сделает код более прозрачным и понятным; вы сможете предъявить основу кода, а потом заняться проработкой деталей.

3.4. Напишите класс MyQueue, который реализует очередь с использованием двух стеков.

С учетом главного различия между очередью и стеком (FIFO против LIFO) нужно изменить методы `peek()` и `pop()` так, чтобы они работали в обратном порядке. Для обращения порядка элементов можно воспользоваться вторым стеком (выталкиваем `s1` и помещаем все элементы в `s2`). В такой реализации каждая операция `peek()` или `pop()` приводит к извлечению всех элементов из `s1` в `s2`, после чего выполняется операция `peek/pop`, а затем все возвращается обратно (с помощью `push`).

Этот алгоритм будет работать, но при последовательном выполнении операций `pop/peek` будут происходить лишние перемещения элементов. Можно реализовать «отложенный» подход, при котором элементы хранятся в `s2` до того момента, пока нам не понадобится их инвертировать.

В этом случае в `stackNewest` помещаются самые новые элементы (на вершину), а в `stackOldest` — самые старые элементы (тоже на вершину). Когда мы исключаем элемент из очереди, необходимо сначала удалить самый старый элемент, то есть вывести его из `stackOldest`. Если `stackOldest` пуст, то следует передать в этот стек все элементы из `stackNewest` в обратном порядке. При вставке элементы заносятся в `stackNewest`, так как новые элементы всегда находятся на вершине.

Приведенный ниже код реализует данный алгоритм:

```
1  public class MyQueue<T> {
2    Stack<T> stackNewest, stackOldest;
3
```

```
4    public MyQueue() {
5      stackNewest = new Stack<T>();
6      stackOldest = new Stack<T>();
7    }
8
9    public int size() {
10     return stackNewest.size() + stackOldest.size();
11   }
12
13   public void add(T value) {
14     /* Все новейшие элементы находятся на вершине stackNewest */
15     stackNewest.push(value);
16   }
17
18   /* Перемещение элементов из stackNewest в stackOldest. Обычно это
19    * делается для выполнения операций с stackOldest. */
20   private void shiftStacks() {
21     if (stackOldest.isEmpty()) {
22       while (!stackNewest.isEmpty()) {
23         stackOldest.push(stackNewest.pop());
24       }
25     }
26   }
27
28   public T peek() {
29     shiftStacks(); // Перенести текущие элементы в stackOldest
30     return stackOldest.peek(); // Получить самый старый элемент.
31   }
32
33   public T remove() {
34     shiftStacks(); // Перенести текущие элементы в stackOldest
35     return stackOldest.pop(); // Извлечь самый старый элемент.
36   }
37 }
```

Во время собеседования вы можете обнаружить, что забыли точные сигнатуры некоторых вызовов API. Не переживайте! Обычно интервьюеры помогают вспомнить подробности. В первую очередь их интересует ваше понимание задачи в целом.

3.5. Напишите программу сортировки стека, в результате которой наименьший элемент оказывается на вершине стека. Вы можете использовать дополнительный временный стек, но элементы не должны копироваться в другие структуры данных (например, в массив). Стек должен поддерживать следующие операции: push, pop, peek, isEmpty.

РЕШЕНИЕ

Одно из возможных решений — реализация примитивного алгоритма сортировки. Мы перебираем весь стек в поисках максимального элемента, извлекаем его и помещаем в новый стек. Находим следующий максимальный элемент и выталкиваем его в новый стек. Такое решение потребует трех стеков: `s1` — исходный; `s2` — окончательный отсортированный стек; `s3` — буфер, используемый при поиске в `s1`. Каждый раз при поиске максимума мы должны извлечь элемент из `s1` и поместить его в `s3`.

К сожалению, для этого потребуется два лишних стека, а по условиям задачи разрешен только один. Можно ли усовершенствовать решение? Да!

Вместо многократного поиска минимума можно отсортировать `s1`, вставляя элементы в нужном порядке в `s2`. Как это реализовать?

Допустим, что у нас есть следующие стеки (стек `s2` «отсортирован», а `s1` нет):

s1	s2
	12
5	8
10	3
7	1

При извлечении элемента 5 из `s1` необходимо найти в `s2` правильное место для вставки. В данном случае оно находится над элементом 3. Как туда попасть? Сначала 5 извлекается из `s1` во временную переменную. Потом 12 и 8 перемещаются из `s2` в `s1`, а элемент 5 заносится в `s2`.

Шаг 1

s1	s2
	12
	8
10	3
7	1

`tmp = 5`

->

Шаг 2

s1	s2
8	
12	
10	3
7	1

`tmp = 5`

->

Шаг 3

s1	s2
8	
12	5
10	3
7	1

`tmp = --`

Элементы 8 и 12 все еще находятся в `s1` — это нормально! Для этих двух чисел повторяются те же действия, которые выполнялись с 5; элемент извлекается с вершины `s1` и помещается в «правильное» место `s2`. (Конечно, поскольку 8 и 12 были перемещены из `s2` в `s1` именно *потому*, что они были больше 5, «правильное» место для этих элементов находится над 5. Возиться с другими элементами `s2` не нужно, и внутренняя часть следующего цикла `while` не будет выполняться, когда `tmp` содержит 8 или 12.)

```
1  Stack<Integer> sort(Stack<Integer> s) {
2    Stack<Integer> r = new Stack<Integer>();
3    while (!s.isEmpty()) {
4      int tmp = s.pop(); // Шаг 1
5      while (!r.isEmpty() && r.peek() > tmp) { // Шаг 2
6        s.push(r.pop());
7      }
8      r.push(tmp); // Шаг 3
9    }
10    return r;
11 }
```

Алгоритм выполняется за время $O(N^2)$ и с затратами памяти $O(N)$.

Если бы было разрешено использовать стеки без ограничений, мы могли бы реализовать модифицированную версию быстрой сортировки или сортировки слиянием.

В решении с сортировкой слиянием создаются два дополнительных стека, а стек делится на две части. Обе части подвергаются рекурсивной сортировке, после чего объединяются в отсортированном порядке в исходном стеке. Обратите внимание: такое решение требует создания двух дополнительных стеков на каждый уровень рекурсии.

В решении с быстрой сортировкой создаются два дополнительных стека, а стек делится на две части по пороговому элементу. Обе части подвергаются рекурсивной сортировке, после чего объединяются в отсортированном порядке в исходном стеке. Это решение, как и предыдущее, требует создания двух дополнительных стеков на каждый уровень рекурсии.

3.6. В приюте для животных есть только собаки и кошки, а работа осуществляется в порядке очереди. Люди должны каждый раз забирать «самое старое» (по времени пребывания в питомнике) животное, но могут выбрать кошку или собаку (животное в любом случае будет «самым старым»). Нельзя выбрать любое понравившееся животное. Создайте структуру данных, которая обеспечивает функционирование этой системы и реализует операции enqueue, dequeueAny, dequeueDog и dequeueCat. Разрешается использование встроенной структуры данных `LinkedList`.

РЕШЕНИЕ

У этой задачи есть много разных решений. Например, можно работать с одной очередью; тогда реализация **dequeueAny** получится простой, но методам **dequeueDog** и **dequeueCat** придется просматривать всю очередь, чтобы найти первую собаку или кошку. Решение получается трудоемким и малоэффективным.

Альтернативный подход прост и понятен: использовать разные очереди для собак и для кошек и поместить их в класс-обертку **AnimalQueue**. Кроме того, нужно хранить временную метку, чтобы знать, когда животное было поставлено в очередь. При вызове **dequeueAny** следует пройти по очередям **dog** и **cat** и найти животное, которое дольше всех находится в очереди.

```
1  abstract class Animal {
2    private int order;
3    protected String name;
4    public Animal(String n) { name = n; }
5    public void setOrder(int ord) { order = ord; }
6    public int getOrder() {  return order; }
7
8    /* Сравнение для получения самого старого элемента. */
9    public boolean isOlderThan(Animal a) {
10      return this.order < a.getOrder();
11   }
12 }
13
14 class AnimalQueue {
15   LinkedList<Dog> dogs = new LinkedList<Dog>();
```

```
16    LinkedList<Cat> cats = new LinkedList<Cat>();
17    private int order = 0; // Используется в качестве временной метки
18
19    public void enqueue(Animal a) {
20      /* Порядок вставки выполняет функции временной метки, что позволяет
21       * сравнивать порядок вставки собак и кошек. */
22      a.setOrder(order);
23      order++;
24
25      if (a instanceof Dog) dogs.addLast((Dog) a);
26      else if (a instanceof Cat) cats.addLast((Cat)a);
27    }
28
29    public Animal dequeueAny() {
30      /* Проверка вершин очередей собак и кошек и извлечение из очереди
31       * элемента с более старым значением. */
32      if (dogs.size() == 0) {
33        return dequeueCats();
34      } else if (cats.size() == 0) {
35        return dequeueDogs();
36     }
37
38      Dog dog = dogs.peek();
39      Cat cat = cats.peek();
40      if (dog.isOlderThan(cat)) {
41        return dequeueDogs();
42      } else {
43        return dequeueCats();
44     }
45    }
46
47    public Dog dequeueDogs() {
48      return dogs.poll();
49    }
50
51    public Cat dequeueCats() {
52      return cats.poll();
53    }
54  }
55
56  public class Dog extends Animal {
57    public Dog(String n) { super(n); }
58  }
59
60  public class Cat extends Animal {
61    public Cat(String n) { super(n); }
62  }
```

Важно, что `Dog` и `Cat` наследуют от класса `Animal`, потому что метод `dequeueAny()` должен быть способен вернуть как объекты `Dog`, так и `Cat`.

При желании порядок вставки можно было бы заменить полноценной временной меткой с датой и временем. Преимущество такого решения в том, что нам придется задавать и поддерживать числовой порядок. Если в списке вдруг появятся два животных с одинаковыми временными метками, то самого старого животного нет (по определению), и вернуть можно любого кандидата.

4

Деревья и графы

4.1. Для заданного направленного графа разработайте алгоритм, проверяющий существование маршрута между двумя узлами.

РЕШЕНИЕ

Данная задача может быть решена простым обходом графа, например поиском в глубину или в ширину. Поиск начинается с одного из двух узлов, и во время обхода выполняется проверка, найден ли другой узел. В процессе выполнения алгоритма следует помечать каждый узел как посещенный, чтобы избежать циклов и повторных посещений узлов.

Приведенный ниже код предоставляет итеративную реализацию обхода графа в ширину:

```
1  enum State { Unvisited, Visited, Visiting; }
2
3  boolean search(Graph g, Node start, Node end) {
4    if (start == end) return true;
5
6    // Работает как очередь
7    LinkedList<Node> q = new LinkedList<Node>();
8
9    for (Node u : g.getNodes()) {
10     u.state = State.Unvisited;
11   }
12   start.state = State.Visiting;
13   q.add(start);
14   Node u;
15   while (!q.isEmpty()) {
16     u = q.removeFirst(); // то есть dequeue()
17     if (u != null) {
18       for (Node v : u.getAdjacent()) {
19         if (v.state == State.Unvisited) {
20           if (v == end) {
21             return true;
22       } else {
23             v.state = State.Visiting;
24             q.add(v);
25       }
26      }
27     }
28       u.state = State.Visited;
29    }
30   }
```

```
31    return false;
32  }
```

Возможно, вам стоит обсудить с интервьюером достоинства и недостатки поиска в глубину и в ширину в этой и других задачах. Поиск в глубину проще реализуется, поскольку может быть выполнен простой рекурсией. Поиск в ширину хорошо подходит для нахождения кратчайших путей, поскольку поиск в глубину может зайти слишком глубоко по графу, прежде чем перейти к прямым соседям.

4.2. Напишите алгоритм создания бинарного дерева поиска с минимальной высотой для отсортированного (по возрастанию) массива с уникальными целочисленными элементами.

РЕШЕНИЕ

Чтобы создаваемое дерево имело минимальную высоту, количество узлов левого и правого поддеревьев должны максимально (насколько это возможно) приближаться друг к другу. Это означает, что корень дерева должен располагаться в середине массива, потому что одна половина элементов должна быть меньше корня, а другая половина — больше.

Построим дерево по этому принципу. Середина каждого подраздела массива становится корнем узла. Левая половина массива превращается в левое поддерево, а правая — в правое поддерево.

При этом можно использовать простой метод `root.insertNode(int v)`, который вставляет значение `v` с помощью рекурсивного процесса, начинающегося с корневого узла. В результате полученное дерево действительно будет иметь минимальную высоту, но эффективность этого алгоритма будет невысока. Каждая вставка потребует обхода дерева, в итоге оценка времени работы составит $O(N \log N)$.

Можно действовать иначе: исключить дополнительные обходы, рекурсивно используя метод `createMinimalBST`. Данный метод получает только фрагмент массива и возвращает его корень минимального дерева.

Алгоритм выглядит так:

1. Вставить средний элемент массива в дерево.
2. Вставить элементы левого подмассива (в левое поддерево).
3. Вставить элементы правого подмассива (в правое поддерево).
4. Повторить рекурсивно.

Ниже приведена реализация этого алгоритма.

```
1  TreeNode createMinimalBST(int array[]) {
2    return createMinimalBST(array, 0, array.length - 1);
3  }
4
5  TreeNode createMinimalBST(int arr[], int start, int end) {
6    if (end < start) {
7      return null;
8    }
9    int mid = (start + end) / 2;
10    TreeNode n = new TreeNode(arr[mid]);
```

```
11    n.left = createMinimalBST(arr, start, mid - 1);
12    n.right = createMinimalBST(arr, mid + 1, end);
13    return n;
14  }
```

Хотя этот код не кажется сложным, в нем легко допустить ошибку со смещением на 1. Тщательно протестируйте все его части.

4.3. Для бинарного дерева разработайте алгоритм, который создает связный список всех узлов, находящихся на каждой глубине (для дерева с глубиной *D* должно получиться *D* связных списков).

РЕШЕНИЕ

На первый взгляд такая задача предполагает обход уровня за уровнем, но на самом деле так поступать не обязательно. Граф можно обходить любым удобным способом, при условии, что в процессе обхода вы постоянно знаете текущий уровень глубины.

Реализуем простую модификацию алгоритма обхода, передающую `level+1` следующему рекурсивному вызову. Приведенный ниже код реализует обход с помощью поиска в глубину:

```
1  void createLevelLinkedList(TreeNode root, ArrayList<LinkedList<TreeNode>> lists,
2                             int level) {
3    if (root == null) return; // Базовый случай
4
5    LinkedList<TreeNode> list = null;
6    if (lists.size() == level) { // Уровень не содержится в списке
7      list = new LinkedList<TreeNode>();
8      /* Уровни всегда обходятся по порядку. Следовательно, при первом
9       * посещении уровня i уровни с 0 по i-1 уже должны быть просмотрены,
10      * поэтому новый уровень можно безопасно добавить в конец. */
11     lists.add(list);
12   } else {
13     list = lists.get(level);
14   }
15   list.add(root);
16   createLevelLinkedList(root.left, lists, level + 1);
17   createLevelLinkedList(root.right, lists, level + 1);
18 }
19
20 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
21   ArrayList<LinkedList<TreeNode>> lists = new ArrayList<LinkedList<TreeNo
de>>();
22   createLevelLinkedList(root, lists, 0);
23   return lists;
24 }
```

Также возможно реализовать модифицированный вариант поиска в ширину. В этой реализации мы начинаем с корня, затем переходим на уровень 2, потом на уровень 3 и т. д.

Таким образом, на каждом уровне `i` все узлы на уровне `i-1` уже будут посещены. а следовательно, чтобы узнать, какие узлы находятся на уровне `i`, можно просто посмотреть на дочерние узлы уровня `i-1`.

Приведенный ниже код реализует данный алгоритм:

```
1  ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
2    ArrayList<LinkedList<TreeNode>> result = new ArrayList<LinkedList<TreeNo
de>>();
3    /* "Посещение" корня */
4    LinkedList<TreeNode> current = new LinkedList<TreeNode>();
5    if (root != null) {
6      current.add(root);
7    }
8
9    while (current.size() > 0) {
10     result.add(current); // Добавление предыдущего уровня
11     LinkedList<TreeNode> parents = current; // Переход на следующий уровень
12     current = new LinkedList<TreeNode>();
13     for (TreeNode parent : parents) {
14       /* Посещение дочерних узлов */
15       if (parent.left != null) {
16         current.add(parent.left);
17     }
18       if (parent.right != null) {
19         current.add(parent.right);
20     }
21    }
22   }
23   return result;
24 }
```

Какое из решений более эффективно? Оба алгоритма выполняются за время $O(N)$, но как насчет памяти? На первый взгляд второе решение более эффективно.

В каком-то смысле это правильно. Первое решение использует $O(\log N)$ рекурсивных вызовов (в сбалансированном дереве), и каждый вызов добавляет новый уровень стека. Второе, итеративное решение, не требует дополнительного пространства.

Однако оба решения возвращают $O(N)$ данных. Дополнительная память $O(\log N)$, расходуемая на рекурсивные вызовы, существенно меньше $O(N)$ — памяти, расходуемой на возвращение данных. Таким образом, в O-записи оба решения одинаково эффективны.

4.4. Реализуйте функцию, проверяющую сбалансированность бинарного дерева. Будем считать дерево сбалансированным, если разница высот двух поддеревьев любого узла не превышает 1.

РЕШЕНИЕ

С этой задачей нам повезло — точно сформулировано, что имеется в виду под сбалансированностью дерева: для каждого узла два поддерева по высоте отличаются не более чем на один узел. Можно реализовать решение, основанное на этом

определении. Для этого достаточно рекурсивно обойти дерево и вычислить высоты каждого поддерева.

```
1  int getHeight(TreeNode root) {
2    if (root == null) return -1; // Базовый случай
3    return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
4  }
5
6  boolean isBalanced(TreeNode root) {
7    if (root == null) return true; // Базовый случай
8
9    int heightDiff = getHeight(root.left) - getHeight(root.right);
10    if (Math.abs(heightDiff) > 1) {
11      return false;
12    } else { // Рекурсия
13      return isBalanced(root.left) && isBalanced(root.right);
14    }
15  }
```

Такое решение работает, но его нельзя назвать эффективным. Для каждого узла приходится рекурсивно исследовать все поддеревья. Это означает, что `getHeight` будет повторно вызываться для тех же узлов. Оценка времени работы алгоритма составит $O(N \log N)$, так как к каждому узлу происходит одно обращение на каждый узел, находящийся выше него.

Хотелось бы избавиться от лишних вызовов `getHeight`.

Если внимательно посмотреть на `getHeight`, то станет ясно, что он может сразу проверять сбалансированность дерева одновременно с проверкой высот. Что делать, если обнаружится, что поддерево не сбалансированно? Просто вернуть код ошибки.

Улучшенный алгоритм проверяет высоту каждого поддерева в ходе рекурсии от вершины. Для каждого узла мы рекурсивно получаем высоту левого и правого поддеревьев с помощью метода `checkHeight`. Этот метод возвращает высоту поддерева, если оно сбалансированно, и код ошибки в противном случае. При получении кода ошибки рекурсия прерывается.

Какое значение использовать в качестве кода ошибки? Высота пустого дерева обычно определяется равной -1, поэтому выбирать это значение нежелательно. Вместо этого мы будем использовать `Integer.MIN_VALUE`.

Приведенный ниже код реализует данный алгоритм:

```
1  int checkHeight(TreeNode root) {
2    if (root == null) return -1;
3
4    int leftHeight = checkHeight(root.left);
5    if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Ошибка
6
7    int rightHeight = checkHeight(root.right);
8    if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Ошибка
9
10    int heightDiff = leftHeight - rightHeight;
11    if (Math.abs(heightDiff) > 1) {
12      return Integer.MIN_VALUE; // Обнаружена ошибка -> вернуть ее
13    } else {
```

```
14      return Math.max(leftHeight, rightHeight) + 1;
15    }
16  }
17
18  boolean isBalanced(TreeNode root) {
19    return checkHeight(root) != Integer.MIN_VALUE;
20  }
```

Этот код выполняется за время $O(N)$ и с затратами памяти $O(\log N)$.

4.5. Реализуйте функцию для проверки того, является ли бинарное дерево бинарным деревом поиска.

РЕШЕНИЕ

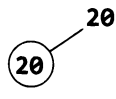
Эту задачу можно решить двумя способами. Первый способ основан на симметричном обходе графа, а второй — на использовании свойства сбалансированного графа: `left <= current < right`.

Решение 1. Симметричный обход

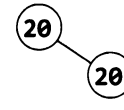
Первое, что приходит в голову, — скопировать при обходе графа все элементы в массив, а затем проверить, отсортирован массив или нет. Такое решение потребует немного дополнительной памяти, но оно работает в большинстве случаев.

Единственная проблема заключается в том, что это решение не справляется с дубликатами. Например, алгоритм не может различить два дерева, изображенных ниже (одно из которых является недействительным), потому что эти деревья имеют одинаковый симметричный обход.

Действительное бинарное дерево поиска



Недействительное бинарное дерево поиска



Впрочем, если предположить, что дерево не содержит дубликатов, такой метод работает. Псевдокод этого метода выглядит так:

```
1  int index = 0;
2  void copyBST(TreeNode root, int[] array) {
3    if (root == null) return;
4    copyBST(root.left, array);
5    array[index] = root.data;
6    index++;
7    copyBST(root.right, array);
8  }
9
10 boolean checkBST(TreeNode root) {
11   int[] array = new int[root.size];
12   copyBST(root, array);
13   for (int i = 1; i < array.length; i++) {
14     if (array[i] <= array[i - 1]) return false;
15   }
```

```
16    return true;
17  }
```

Обратите внимание на необходимость отслеживания логического «конца» массива, так как память под него выделяется для хранения всех элементов графа.

Если проанализировать это решение, то становится ясно, что на самом деле массив не нужен. Он ни разу не используется ни для чего, кроме сравнения элемента с предыдущим. Тогда почему бы просто не отслеживать значение последнего элемента и сравнивать его с текущим?

Ниже приведена модифицированная версия алгоритма:

```
1  Integer last_printed = null;
2  boolean checkBST(TreeNode n) {
3    if (n == null) return true;
4
5    // Проверка / рекурсия левого поддерева
6    if (!checkBST(n.left)) return false;
7
8    // Проверка текущего узла
9    if (last_printed != null && n.data <= last_printed) {
10     return false;
11   }
12   last_printed = n.data;
13
14   // Проверка / рекурсия правого поддерева
15   if (!checkBST(n.right)) return false;
16
17   return true; // Все хорошо!
18 }
```

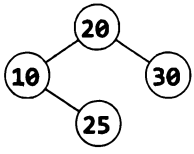
Мы использовали `Integer` вместо `int`, чтобы узнать о присваивании значения `last_printed`.

Если вам не нравится идея использования статических переменных, можно изменить код, добавив класс-обертку для целочисленного значения:

```
1  class WrapInt {
2    public int value;
3  }
```

Или если вы пишете код на C++ или другом языке, который поддерживает передачу целых чисел по ссылке, просто воспользуйтесь этой возможностью.

Решение 2. Использование минимумов/максимумов



Во втором варианте решения используется определение бинарного дерева поиска.

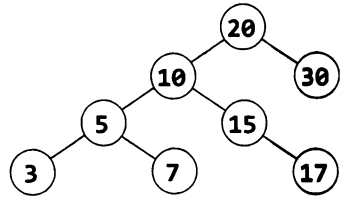
Что такое бинарное дерево поиска? Мы знаем, что каждый узел должен соответствовать условию `left.data <= current.data < right.data`, но этого недостаточно. Рассмотрим маленькое дерево.

Хотя каждый узел этого дерева больше левого и меньше правого, это дерево нельзя назвать бинарным деревом поиска, поскольку 25 находится не на своем месте.

Более точно условие формулируется так: все левые узлы должны быть меньше или равны текущему узлу, который, в свою очередь, должен быть меньше всех узлов справа.

Используя это условие, можно решить задачу, передавая минимальные и максимальные значения. При переборе узлов дерева проверка осуществляется по последовательно сужающимся диапазонам.

Рассмотрим следующее дерево:



Начнем с диапазона (`min = NULL, max = NULL`), который обязательно содержит корень (`NULL` означает, что минимум или максимум отсутствует). Затем двигаемся влево, проверяя все узлы в интервале (`min = NULL, max = 20`). Затем двигаемся в правую сторону и проверяем узлы в диапазоне (`min = 20, max = NULL`).

Используя этот алгоритм, можно обойти все дерево. При движении влево обновляется максимум, при движении вправо обновляется минимум. Если проверка не пройдена, возвращается `false`.

Временная сложность этого решения — $O(N)$, где N — количество узлов в дереве. Можно доказать, что улучшить это решение невозможно, так как любой алгоритм должен проверять все N узлов.

Из-за использования рекурсии пространственная сложность алгоритма на сбалансированном дереве составит $O(\log N)$. Нам понадобится $O(\log N)$ рекурсивных вызовов в стеке, чтобы добраться до максимальной глубины дерева.

Рекурсивный код данного алгоритма выглядит так:

```
1  boolean checkBST(TreeNode n) {
2    return checkBST(n, null, null);
3  }
4
5  boolean checkBST(TreeNode n, Integer min, Integer max) {
6    if (n == null) {
7      return true;
8    }
9    if ((min != null && n.data <= min) || (max != null && n.data > max)) {
10     return false;
11   }
12
13   if (!checkBST(n.left, min, n.data) || !checkBST(n.right, n.data, max)) {
14     return false;
15   }
16   return true;
17 }
```

Не забывайте, что в рекурсивных алгоритмах всегда нужно обеспечивать корректную обработку базовых случаев, а также случаев с `null`.

4.6. Напишите алгоритм поиска «следующего» узла для заданного узла в бинарном дереве поиска. Считайте, что у каждого узла есть ссылка на его родителя.

РЕШЕНИЕ

Вспомните, что при симметричном обходе графа сначала обрабатывается левое поддерево, затем текущий узел и только после него — правое поддерево. Чтобы взяться за эту задачу, необходимо очень внимательно подумать о том, что же происходит.

Возьмем некоторый узел графа. Порядок обхода нам известен: левое поддерево, текущий узел, правое поддерево. Таким образом, следующий узел, который мы посетим, должен находиться справа.

Но какой именно узел правого поддерева? Это должен быть первый узел, который мы посетили бы при симметричном обходе этого поддерева. Таким образом, нам нужен крайний левый узел правого поддерева.

Но что делать, если у узла нет правого поддерева? Ситуация усложняется.

Если у узла `n` нет правого поддерева, обход поддерева `n` на этом заканчивается. Следует вернуться к предку `n` (назовем его `q`).

Если `n` находится в левом поддереве `q`, то `q` станет следующим узлом, который мы должны посетить (порядок обхода: `левый -> текущий -> правый`).

Если `n` находился правее `q`, то это означает, что мы полностью обошли поддерево `q`. Теперь нужно двигаться от `q` вверх, пока не найдется узел `x`, который мы еще не обследовали полностью. Как мы узнаем, что узел `x` не обследован полностью? Это происходит при переходе от левого узла к его родителю. Левый узел полностью обследован, а его родитель — нет.

Псевдокод выглядит примерно так:

```
1  Node inorderSucc(Node n) {
2    if (у n есть правое поддерево) {
3      return крайний левый дочерний узел правого поддерева
4    } else {
5      while (n является правым дочерним узлом n.parent) {
6        n = n.parent; // Подняться
7     }
8      return n.parent; // Родитель еще не обойден
9    }
10 }
```

А что произойдет, если мы обойдем дерево полностью раньше, чем найдем левый дочерний элемент? Такая ситуация возможна, только если мы закончили симметричный обход. Иначе говоря, если мы *уже* находимся на правом краю дерева, то вернуть следующий элемент не удастся, и следует возвратить `null`.

Следующий код реализует этот алгоритм (и правильно обрабатывает случай `null`):

```
1  TreeNode inorderSucc(TreeNode n) {
2    if (n == null) return null;
3    /* Найден правый дочерний узел -> вернуть крайний левый узел
```

```
    * правого поддерева. */
   if (n.right != null) {
     return leftMostChild(n.right);
   } else {
     TreeNode q = n;
     TreeNode x = q.parent;
      // Подниматься наверх, пока не окажемся слева
     while (x != null && x.left != q) {
       q = x;
       x = x.parent;
     }
     return x;
   }
 }

 TreeNode leftMostChild(TreeNode n) {
   if (n == null) {
     return null;
   }
   while (n.left != null) {
     n = n.left;
   }
   return n;
 }
```

Это не самая сложная задача с алгоритмической точки зрения, но написать безупречный код для ее решения достаточно трудно. В таких случаях полезно использовать псевдокод, чтобы тщательно обозначить все возможные случаи.

4.7. Имеется список проектов и список зависимостей (список пар проектов, для которых первый проект зависит от второго проекта). Проект может быть построен только после построения всех его зависимостей. Найдите такой порядок построения, который позволит построить все проекты. Если действительного порядка не существует, верните признак ошибки.

Пример:

Ввод:

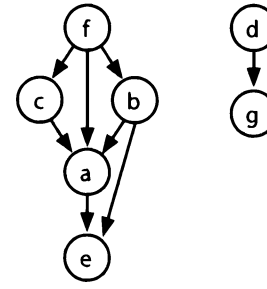
```
проекты: a, b, c, d, e, f
зависимости: (d, a), (b, f), (d, b), (a, f), (c, d)
```

Вывод:

```
f, e, a, b, d, c
```

РЕШЕНИЕ

Вероятно, информацию лучше всего наглядно представить в виде графа. Будьте внимательны с направлением стрелок. На приведенном ниже графе стрелка от d к g означает, что проект d должен быть завершен ранее g. Также стрелки на схеме можно провести в обратном направлении, но вы должны действовать последовательно и четко обозначить смысл диаграммы.



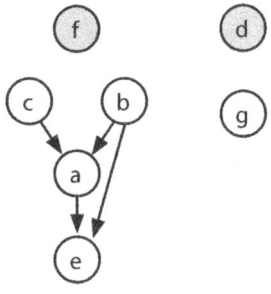
При создании этого примера я стремилась добиться следующих целей:

- Узлы должны быть помечены несколько хаотично. Если бы узел *a* располагался наверху, ниже следовали дочерние узлы *b* и *c*, а за ними *d* и *e*, это бы создавало ошибочное впечатление, будто алфавитный порядок соответствует порядку построения.
- Граф должен был состоять из нескольких частей/компонентов, поскольку связный граф скорее является частным случаем.
- В графе должен был присутствовать узел, соединенный с узлом, который не может следовать непосредственно после него. Например, узел *f* соединен с *a*, но а не может следовать сразу же после *f* (потому что узлы *b* и *c* должны быть построены после *a*, но до *f*).
- Граф должен быть достаточно большим, чтобы на нем можно было выявить возникающую закономерность.
- Узлы должны были иметь несколько зависимостей.

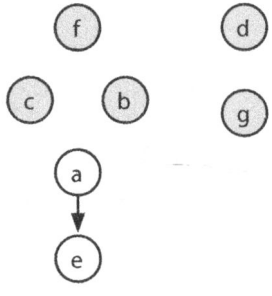
Итак, теперь у нас имеется хороший пример, и мы можем взяться за алгоритм.

Решение 1

С чего начать? Существуют ли проекты, которые заведомо могут быть построены немедленно?



Порядок завершения: f, d, c, b, g



Да, существуют. Узлы, не имеющие входящих ребер, могут быть построены немедленно, потому что они ни от чего не зависят. Добавим все такие узлы в порядок построения. В приведенном примере это означает порядок `f`, `d` (или `d`, `f`).

Когда это будет сделано, зависимости некоторых узлов от `d` и `f` становятся несущественными, потому что `d` и `f` уже построены. Это новое состояние можно представить, удалив из графа исходящие ребра `d` и `f`.

Теперь мы знаем, что проекты `c`, `b` и `g` можно построить без зависимостей, потому что они не имеют входящих ребер. Построим эти проекты и удалим исходящие из них ребра.

На следующем шаге можно построить проект *a*; сделаем это и удалим исходящие из него ребра. Остается только узел e. После его завершения мы получаем следующий порядок построения проектов:

```
f, d, c, b, g, a, e
```

Этот алгоритм работает всегда, или нам просто повезло? Поразмыслите над логикой происходящего.

1. Сначала добавляются узлы, не имеющие входящих ребер. Если множество проектов может быть построено, должен существовать некий «первый» проект, который не может иметь никаких зависимостей. Если проект не имеет зависимостей (входящих ребер), безусловно, его первоочередное построение ничему не повредит.
2. Затем были удалены все ребра, выходящие из корневых узлов. Это разумно — после того, как корневые проекты будут построены, становится неважно, зависит ли от них другой проект.
3. После этого были найдены узлы, не имеющие входящих ребер. Применяя логику шагов 1 и 2, мы видим, что их построение ничему не повредит. Теперь можно повторить те же действия: найти узлы, не имеющие зависимостей, добавить их в порядок построения, удалить исходящие ребра и повторить.
4. Что, если в графе остались узлы, но все они имеют зависимости (входящие ребра)? Это означает, что построить систему невозможно и нужно вернуть код ошибки.

Реализация очень близко воспроизводит этот алгоритм.

Инициализация и подготовка:

1. Построить граф, в котором каждый проект представлен узлом, а исходящие ребра представляют зависимости. Иначе говоря, если из узла A выходит ребро в B (A->B), это означает, что B зависит от A, а значит, проект A должен быть построен раньше B. Каждый узел также отслеживает количество *входящих* ребер.
2. Инициализировать массив `buildOrder`. После того как для проекта будет определен порядок построения, он добавляется в массив. Также при продолжении перебора массива указатель `toBeProcessed` используется для ссылки на следующий узел, подлежащий полной обработке.
3. Найти все узлы с нулем входящих ребер и добавить их в массив `buildOrder`. Установить указатель `toBeProcessed` в начало массива.

Повторять до тех пор, пока `toBeProcessed` не окажется в конце `buildOrder`:

1. Прочитать узел по указателю `toBeProcessed`.
   - Если узел равен `null`, значит, все оставшиеся узлы содержат зависимость и мы обнаружили цикл.
2. Для каждого дочернего узла:
   - Уменьшить `child.dependencies` (количество входящих ребер).
   - Если значение `child.dependencies` равно нулю, добавить дочерний узел в конец `buildOrder`.
3. Увеличить `toBeProcessed`.

Ниже приведена реализация этого алгоритма.

```
1  /* Определение правильного порядка построения. */
2  Project[] findBuildOrder(String[] projects, String[][] dependencies) {
3    Graph graph = buildGraph(projects, dependencies);
4    return orderProjects(graph.getNodes());
5  }
6
```

```
/* Построить граф с добавлением ребра (a, b), если b зависит от a.
 * Предполагается, что зависимость (a, b) означает, что b зависит
 * от a, и проект a должен быть построен перед b. */
Graph buildGraph(String[] projects, String[][] dependencies) {
  Graph graph = new Graph();
  for (String project : projects) {
    graph.createNode(project);
  }

  for (String[] dependency : dependencies) {
    String first = dependency[0];
    String second = dependency[1];
    graph.addEdge(first, second);
  }

  return graph;
}

/* Получение списка проектов в правильном порядке построения. */
Project[] orderProjects(ArrayList<Project> projects) {
  Project[] order = new Project[projects.size()];

  /* Сначала добавляются "корни".*/
  int endOfList = addNonDependent(order, projects, 0);

  int toBeProcessed = 0;
  while (toBeProcessed < order.length) {
    Project current = order[toBeProcessed];

    /* Имеется циклическая зависимость, так как не осталось
     * проектов с нулем зависимостей. */
    if (current == null) {
      return null;
   }

    /* Удаление зависимостей от текущего узла. */
    ArrayList<Project> children = current.getChildren();
    for (Project child : children) {
      child.decrementDependencies();
   }

    /* Добавление дочерних узлов без зависимостей. */
    endOfList = addNonDependent(order, children, endOfList);
    toBeProcessed++;
  }

  return order;
}

/* Вспомогательная функция для вставки проектов с нулем зависимостей
 * в массив, начиная с индекса offset. */
int addNonDependent(Project[] order, ArrayList<Project> projects, int offset) {
  for (Project project : projects) {
    if (project.getNumberDependencies() == 0) {
      order[offset] = project;
      offset++;
   }
```

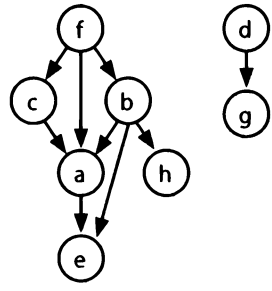
```
64    }
65    return offset;
66  }
67
68  public class Graph {
69    private ArrayList<Project> nodes = new ArrayList<Project>();
70    private HashMap<String, Project> map = new HashMap<String, Project>();
71
72    public Project getOrCreateNode(String name) {
73      if (!map.containsKey(name)) {
74        Project node = new Project(name);
75        nodes.add(node);
76        map.put(name, node);
77      }
78
79      return map.get(name);
80    }
81
82    public void addEdge(String startName, String endName) {
83      Project start = getOrCreateNode(startName);
84      Project end = getOrCreateNode(endName);
85      start.addNeighbor(end);
86    }
87
88    public ArrayList<Project> getNodes() { return nodes; }
89  }
90
91  public class Project {
92    private ArrayList<Project> children = new ArrayList<Project>();
93    private HashMap<String, Project> map = new HashMap<String, Project>();
94    private String name;
95    private int dependencies = 0;
96
97    public Project(String n) { name = n; }
98
99    public void addNeighbor(Project node) {
100     if (!map.containsKey(node.getName())) {
101       children.add(node);
102       node.incrementDependencies();
103     }
104   }
105
106   public void incrementDependencies() { dependencies++; }
107   public void decrementDependencies() { dependencies--; }
108
109   public String getName() { return name; }
110   public ArrayList<Project> getChildren() { return children; }
111   public int getNumberDependencies() { return dependencies; }
112 }
```

Решение выполняется за время $O(P + D)$, где P — количество проектов, а D — количество пар зависимостей.

Примечание: возможно, вы узнаете в этом листинге алгоритм топологической сортировки на с. 668. Мы вывели его с нуля. Большинству людей этот алгоритм неизвестен, и интервьюер ожидает, что вы сможете вывести его самостоятельно.

Решение 2

Также для нахождения пути построения можно воспользоваться алгоритмом поиска в глубину (DFS).

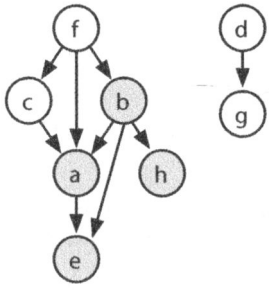


Допустим, мы выбираем произвольный узел (скажем, *b*) и проводим для него поиск в глубину. Когда мы добираемся до конца пути и не можем двигаться дальше (что происходит в узлах *h* и *e*), это означает, что завершающие узлы могут быть последними — другие проекты от них не зависят.

```
DFS(b)                                  // Шаг 1
  DFS(h)                                // Шаг 2
    порядок построения = ..., h         // Шаг 3
  DFS(a)                                // Шаг 4
   DFS(e)                               // Шаг 5
      порядок построения = ..., e, h    // Шаг 6
   ...                                  // Шаг 7+
  ...
```

Теперь разберемся, что происходит в узле *a* при возвращении из DFS узла *e*. Все его зависимости были построены, а это означает, что ничто не препятствует построению *a*. А после построения *a* мы знаем, что все зависимости *b* были построены, и теперь можно построить *b*.

```
DFS(b)                                  // Шаг 1
  DFS(h)                                // Шаг 2
    порядок построения = ..., h         // Шаг 3
  DFS(a)                                // Шаг 4
   DFS(e)                               // Шаг 5
      порядок построения = ..., e, h    // Шаг 6
    порядок построения = ..., a, e, h   // Шаг 7
  DFS(e) -> return                      // Шаг 8
  порядок построения = ..., b, a, e, h  // Шаг 9
```



Пометим эти узлы как уже построенные на тот случай, если другому проекту понадобится построить их.

Что дальше? Можно снова начать с любого старого узла, провести для него поиск в глубину (DFS) и добавить узел в начало очереди построения при завершении поиска в глубину.

```
DFS(d)
  DFS(g)
    порядок построения = ..., g, b, a, e, h
```

```
  порядок построения = ..., d, g, b, a, e, h
DFS(f)
  DFS(c)
    порядок построения = ..., c, d, g, b, a, e, h
  порядок построения = f, c, d, g, b, a, e, h
```

В таких алгоритмах следует продумать проблему циклов. При наличии цикла в графе действительного порядка построения не существует. Тем не менее было бы нежелательно попадать в бесконечный цикл только потому, что решения нет.

Цикл возникает в том случае, если при выполнении поиска в глубину для узла мы возвращаемся на тот же путь. Следовательно, нужен сигнал, который означает: «Этот узел все еще обрабатывается, поэтому если мы встретим его снова — у нас проблемы».

В такой ситуации можно пометить каждый узел как «частично обработанный» непосредственно перед тем, как запускать для него поиск в глубину. Если в процессе поиска будет обнаружен любой узел в частично обработанном состоянии, это указывает на наличие проблемы. При завершении поиска в глубину для узла необходимо обновить его состояние.

Также следует предусмотреть состояние «узел обработан/построен», чтобы алгоритм не пытался построить его заново. Таким образом, узел может находиться в одном из трех состояний: полностью обработанном (`COMPLETED`), частично обработанном (`PARTIAL`) и необработанном (`BLANK`).

Ниже приведена реализация этого алгоритма.

```
1  Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
2    Graph graph = buildGraph(projects, dependencies);
3    return orderProjects(graph.getNodes());
4  }
5
6  Stack<Project> orderProjects(ArrayList<Project> projects) {
7    Stack<Project> stack = new Stack<Project>();
8    for (Project project : projects) {
9      if (project.getState() == Project.State.BLANK) {
10       if (!doDFS(project, stack)) {
11         return null;
12     }
13     }
14   }
15   return stack;
16 }
17
18 boolean doDFS(Project project, Stack<Project> stack) {
19   if (project.getState() == Project.State.PARTIAL) {
20     return false; // Цикл
21   }
22
23   if (project.getState() == Project.State.BLANK) {
24     project.setState(Project.State.PARTIAL);
25     ArrayList<Project> children = project.getChildren();
26     for (Project child : children) {
27       if (!doDFS(child, stack)) {
28         return false;
29     }
```

```
30      }
31       project.setState(Project.State.COMPLETE);
32       stack.push(project);
33     }
34     return true;
35   }
36
37   /* То же, что прежде */
38   Graph buildGraph(String[] projects, String[][] dependencies) {...}
39   public class Graph {}
40
41   /* Фактически эквивалентно предыдущему решению, с добавлением
42    * информации состояния и удалением счетчика зависимостей. */
43   public class Project {
44     public enum State {COMPLETE, PARTIAL, BLANK};
45     private State state = State.BLANK;
46     public State getState() { return state; }
47     public void setState(State st) { state = st; }
48     /* Повторяющийся код опущен для краткости */
49   }
```

Как и более ранний алгоритм, это решение выполняется за время $O(P+D)$, где P — количество проектов, а D — количество пар зависимостей.

Кстати говоря, эта задача называется топологической сортировкой: линейное упорядочение вершин графа таким образом, что для каждого ребра (a, b) a в линейном порядке предшествует b.

4.8. Создайте алгоритм и напишите код поиска первого общего предка двух узлов бинарного дерева. Постарайтесь избежать хранения дополнительных узлов в структуре данных. Примечание: бинарное дерево не обязательно является бинарным деревом поиска.

РЕШЕНИЕ

Если бы речь шла о бинарном дереве поиска, то можно было бы изменить операцию `find` для двух узлов и определить, где расходятся пути. К сожалению, по условиям задачи дерево не является бинарным деревом поиска, поэтому нужно искать другие подходы.

Предположим, мы хотим найти общего предка для узлов `p` и `q`. Необходимо решить один важный вопрос: имеют ли узлы нашего дерева связь с предками.

Решение 1. Со ссылками на предков

Если у каждого узла есть ссылка на предка, можно отследить пути `p` и `q`, вплоть до их пересечения. Фактически задача эквивалентна задаче 2.7, в которой мы искали пересечение двух связных списков. «Связным списком» в данном случае становится путь от каждого узла до корня.

```
1  TreeNode commonAncestor(TreeNode p, TreeNode q) {
2    int delta = depth(p) - depth(q); // Вычисление разности глубин
3    TreeNode first = delta > 0 ? q : p; // Узел с меньшей глубиной
4    TreeNode second = delta > 0 ? p : q; // Узел с большей глубиной
5    second = goUpBy(second, Math.abs(delta)); // Глубокий узел поднимается
```

```
6
7    /* Поиск пересечения путей. */
8    while (first != second && first != null && second != null) {
9      first = first.parent;
10     second = second.parent;
11   }
12   return first == null || second == null ? null : first;
13 }
14
15 TreeNode goUpBy(TreeNode node, int delta) {
16   while (delta > 0 && node != null) {
17     node = node.parent;
18     delta--;
19   }
20   return node;
21 }
22
23 int depth(TreeNode node) {
24   int depth = 0;
25   while (node != null) {
26     node = node.parent;
27     depth++;
28   }
29   return depth;
30 }
```

Решение выполняется за время $O(d)$, где d — глубина более глубокого узла.

Решение 2. Со ссылками на предков (улучшенное время выполнения в худшем случае)

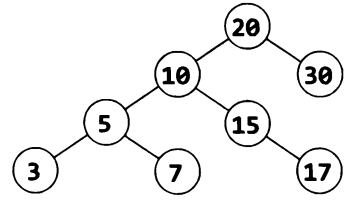
По аналогии с предыдущим решением можно отследить путь p наверх и проверить, накрывает ли какой-либо из узлов этого пути узел q. Первый узел, накрывающий q (мы уже знаем, что каждый узел этого пути накрывает p), должен быть первым общим предком.

Обратите внимание: нам не нужно заново проверять все поддерево. При перемещении от узла x к его родителю y все узлы поддерева x уже были проверены для q. Таким образом, остается проверить только новые «ненакрытые» узлы, которыми являются узлы в поддереве соседа n.

Предположим, мы ищем первого общего предка узла p=7 и узла q=17. Поднимаясь к `p.parent` (5), мы обнаруживаем поддерево с корнем 3. Следовательно, необходимо проверить это поддерево на вхождение q.

Затем мы переходим к узлу 10, открывая поддерево с корнем 15. Ищем в этом поддереве узел 17 и находим его!

Чтобы реализовать этот алгоритм, можно просто подняться наверх от `p`, сохранив родительский и соседний узлы в переменных `parent` и `sibling`. (Соседний узел `sibling` всегда является дочерним по отношению к `parent`.) При каждой итерации `sibling` присваивается соседний узел старого `parent`, а `parent` присваивается `parent.parent`.



```
1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2    /* Проверяет, что узел либо отсутствует в дереве, либо накрывает другой. */
3    if (!covers(root, p) || !covers(root, q)) {
4      return null;
5    } else if (covers(p, q)) {
6      return p;
7    } else if (covers(q, p)) {
8      return q;
9    }
10
11    /* Перемещаться вверх, пока не будет найден узел, накрывающий q. */
12    TreeNode sibling = getSibling(p);
13    TreeNode parent = p.parent;
14    while (!covers(sibling, q)) {
15      sibling = getSibling(parent);
16      parent = parent.parent;
17    }
18    return parent;
19  }
20
21  boolean covers(TreeNode root, TreeNode p) {
22    if (root == null) return false;
23    if (root == p) return true;
24    return covers(root.left, p) || covers(root.right, p);
25  }
26
27  TreeNode getSibling(TreeNode node) {
28    if (node == null || node.parent == null) {
29      return null;
30    }
31
32    TreeNode parent = node.parent;
33    return parent.left == node ? parent.right : parent.left;
34  }
```

Алгоритм выполняется за время $O(t)$, где t — размер поддерева первого общего предка. В худшем случае время выполнения составит $O(n)$, где n — количество узлов в дереве. Чтобы прийти к этому времени выполнения, достаточно заметить, что каждый узел в поддереве просматривается один раз.

Решение 3. Без ссылок на родителей

Также возможно проследить цепочку, в которой p и q находятся на одной стороне. То есть если и p, и q находятся слева от узла, то, чтобы найти общего предка, следует двигаться влево. Если они оба находятся справа, то и двигаться нужно вправо. Если p и q окажутся по разные стороны от узла — ближайший общий предок найден.

Код, приведенный ниже, реализует этот подход:

```
1  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
2    /* Проверка ошибки - один узел отсутствует в дереве. */
3    if (!covers(root, p) || !covers(root, q)) {
4      return null;
5    }
6    return ancestorHelper(root, p, q);
7  }
```

```
8
9  TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
10   if (root == null || root == p || root == q) {
11     return root;
12   }
13
14   boolean pIsOnLeft = covers(root.left, p);
15   boolean qIsOnLeft = covers(root.left, q);
16   if (pIsOnLeft != qIsOnLeft) { // Узлы на разных сторонах
17     return root;
18   }
19   TreeNode childSide = pIsOnLeft ? root.left : root.right;
20   return ancestorHelper(childSide, p, q);
21 }
22
23 boolean covers(TreeNode root, TreeNode p) {
24   if (root == null) return false;
25   if (root == p) return true;
26   return covers(root.left, p) || covers(root.right, p);
27 }
```

Этот алгоритм потребует $O(n)$ времени на сбалансированном дереве, поскольку `covers` вызывается для `2n` узлов при первом вызове (`n` узлов слева и `n` узлов справа). После этого алгоритм выбирает направление (налево или направо), в новой точке `covers` вызывается для `2n/2` узлов, затем для `2n/4` и т. д. В результате общее время выполнения составит $O(n)$.

Казалось бы, лучшего результата в контексте асимптотического времени выполнения добиться уже невозможно — мы должны проанализировать каждый узел в дереве. Однако мы можем уменьшить время на постоянный множитель.

Решение 4. Оптимизация

Хотя решение 2 является оптимальным по времени выполнения, его можно еще немного улучшить, а именно: `covers` ищет все узлы под `root` в поисках `p` или `q`, включая узлы в каждом поддереве (`root.left` и `root.right`). Затем `covers` выбирает одно из поддеревьев и проводит поиск по всем его узлам. Таким образом, каждое поддерево проходится много раз.

Можно заметить, что все дерево достаточно проверить всего один раз для поиска `p` и `q`. Далее результаты поиска можно будет «поднимать» в предшествующие узлы в стеке. Основная логика остается той же, что и в предыдущем решении.

Мы рекурсивно проходимся по всему дереву с помощью функции `commonAncestor (TreeNode root, TreeNode p, TreeNode q)`, которая возвращает следующие значения:

- `p`, если поддерево `root` содержит `p` (и не содержит `q`);
- `q`, если поддерево `root` содержит `q` (и не содержит `p`);
- `null`, если ни `p`, ни `q` нет в поддереве `root`;
- в противном случае возвращается общий предок `p` и `q`.

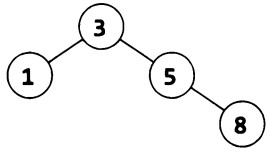
Найти общего предка `p` и `q` в последнем случае несложно. Если и `commonAncestor(n.left, p, q)` и `commonAncestor(n.right, p, q)` возвращают значения, отличные от

null (указывающие на то, что p и q были найдены в разных поддеревьях), n является общим предком для p и q.

Ниже приведена первая версия решения. Код содержит ошибку. Удастся ли вам ее найти?

```
1  /* В следующем коде имеется ошибка. */
2  TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
3    if (root == null) return null;
4    if (root == p && root == q) return root;
5
6    TreeNode x = commonAncestor(root.left, p, q);
7    if (x != null && x != p && x != q) { // Предок уже найден
8      return x;
9    }
10
11    TreeNode y = commonAncestor(root.right, p, q);
12    if (y != null && y != p && y != q) { // Предок уже найден
13      return y;
14    }
15
16    if (x != null && y != null) { // p и q found в разных поддеревьях
17      return root; // Общий предок
18    } else if (root == p || root == q) {
19      return root;
20    } else {
21      return x == null ? y : x; /* Вернуть значение, отличное от null */
22    }
23  }
```

Проблема в этом коде проявляется тогда, когда узел не содержится в дереве. Например, взгляните на следующее дерево:



Предположим, был сделан вызов commonAncestor(node 3, node 5, node 7). Конечно, узла 7 не существует — и здесь-то возникает проблема. Порядок вызовов выглядит так:

```
1  commonAnc(node 3, node 5, node 7)            // --> 5
2    вызов commonAnc(node 1, node 5, node 7)    // --> null
3    вызов commonAnc(node 5, node 5, node 7)    // --> 5
4      вызов commonAnc(node 8, node 5, node 7)  // --> null
```

Другими словами, когда мы вызываем commonAncestor для правого поддерева, код вернет узел 5 (и это правильно). Проблема заключается в следующем: при обнаружении общего предка p и q функция не может различить два случая:

- случай 1: p — дочерний элемент q (или q — дочерний элемент p);
- случай 2: p — присутствует в дереве, а q — нет (и наоборот).

В обоих случаях commonAncestor возвратит p. В первом случае ответ правильный, но во втором случае возвращаемое значение должно быть равно null.

Необходимо разделить эти два случая; приведенный ниже код решает эту проблему, возвращая два значения — узел и флаг, по которому можно понять, является ли узел общим предком.

```
class Result {
  public TreeNode node;
  public boolean isAncestor;
  public Result(TreeNode n, boolean isAnc) {
    node = n;
    isAncestor = isAnc;
  }
}

TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  Result r = commonAncestorHelper(root, p, q);
  if (r.isAncestor) {
    return r.node;
  }
  return null;
}

Result commonAncHelper(TreeNode root, TreeNode p, TreeNode q) {
  if (root == null) return new Result(null, false);

  if (root == p && root == q) {
    return new Result(root, true);
  }

  Result rx = commonAncHelper(root.left, p, q);
  if (rx.isAncestor) { // Найден общий предок
    return rx;
  }

  Result ry = commonAncHelper(root.right, p, q);
  if (ry.isAncestor) { // Найден общий предок
    return ry;
  }

  if (rx.node != null && ry.node != null) {
    return new Result(root, true); // Общий предок
  } else if (root == p || root == q) {
    /* Если текущим узлом является p или q и один из этих узлов
     * также найден в поддереве, то это предок, и флагу присваивается true. */
    boolean isAncestor = rx.node != null || ry.node != null;
    return new Result(root, isAncestor);
  } else {
    return new Result(rx.node!=null ? rx.node : ry.node, false);
  }
}
```

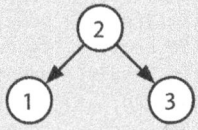
Конечно, раз эта проблема встречается только в том случае, если p или q не присутствует в дереве, возможно альтернативное решение: сначала провести поиск по всему дереву и убедиться в том, что оба узла существуют.

4.9. Бинарное дерево поиска было создано обходом массива слева направо и вставкой каждого элемента. Для заданного бинарного дерева поиска с разными элементами выведите все возможные массивы, которые могли привести к созданию этого дерева.

Пример:

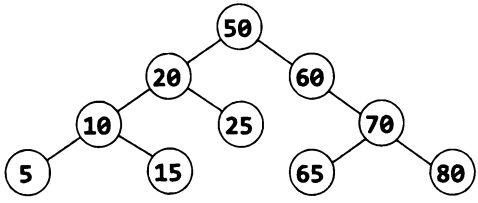
Ввод:

Вывод: `{2, 1, 3}, {2, 3, 1}`



РЕШЕНИЕ

Задачу проще объяснять на хорошем примере:



Также следует подумать об упорядочении элементов в бинарном дереве поиска. Для каждого конкретного узла все узлы слева от него должны быть меньше всех узлов, находящихся справа. Достигнув места, в котором нет узла, мы вставляем новое значение.

Все это означает, что для создания приведенного выше дерева первый элемент массива должен быть равен 50. Если бы он имел любое другое значение, то корнем было бы это значение.

Что еще? Некоторые разработчики приходят к поспешному выводу, что все элементы слева должны быть вставлены перед элементами справа, но на самом деле это не так. Более того, истинно обратное: порядок вставки левых и правых элементов роли не играет.

После достижения значения 50 все элементы, меньшие 50, будут направляться налево, а все элементы, большие 50, будут направляться направо. Какой из двух элементов будет вставлен первым, 60 или 20, роли не играет.

Рассмотрим задачу с точки зрения рекурсии. Если бы у нас в распоряжении были все массивы, которые могли бы создать поддерево с корнем 20 (назовем это множество `arraySet20`), и все массивы, которые могли бы создать поддерево с корнем 60 (назовем это множество `arraySet60`), как это помогло бы прийти к полному ответу? Мы могли бы «переплести» каждый массив из `arraySet20` с каждым массивом из `arraySet60`, а затем добавить перед каждым массивом 50.

Что мы имеем в виду под «переплетением»? Два массива объединяются всеми возможными способами, с сохранением относительного порядка элементов в каждом массиве:

```
массив1: {1, 2}
массив2: {3, 4}
переплетение: {1, 2, 3, 4}, {1, 3, 2, 4}, {1, 3, 4, 2},
    {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 4, 1, 2}
```

Обратите внимание: если в исходных множествах массивов не было дубликатов, нам не придется беспокоиться о том, что дубликаты будут созданы в результате переплетения.

Осталось упомянуть о том, как работает переплетение. Рассмотрим задачу переплетения `{1, 2, 3}` с `{4, 5, 6}` с точки зрения рекурсии. Какие подзадачи можно выделить?

- Вставить 1 перед всеми переплетениями `{2, 3}` и `{4, 5, 6}`.
- Вставить 4 перед всеми переплетениями `{1, 2, 3}` и `{5, 6}`.

Чтобы реализовать рекурсию, мы используем для хранения данных связные списки; это упростит добавление и удаление элементов. При выполнении рекурсии элементы, снабженные префиксом, продвигаются вниз по рекурсивной структуре. Когда значение `first` или `second` становится пустым, мы добавляем остаток к `prefix` и сохраняем результат.

Схема работает примерно так:

```
weave(first, second, prefix):
  weave({1, 2}, {3, 4}, {})
    weave({2}, {3, 4}, {1})
      weave({}, {3, 4}, {1, 2})
        {1, 2, 3, 4}
      weave({2}, {4}, {1, 3})
        weave({}, {4}, {1, 3, 2})
          {1, 3, 2, 4}
        weave({2}, {}, {1, 3, 4})
          {1, 3, 4, 2}
    weave({1, 2}, {4}, {3})
      weave({2}, {4}, {3, 1})
        weave({}, {4}, {3, 1, 2})
          {3, 1, 2, 4}
        weave({2}, {}, {3, 1, 4})
          {3, 1, 4, 2}
      weave({1, 2}, {}, {3, 4})
        {3, 4, 1, 2}
```

Теперь рассмотрим реализацию удаления, скажем, 1 из `{1, 2}` с последующей рекурсией. Мы должны проявить осторожность с модификацией списка, так как более позднему рекурсивному вызову (например, `weave({1, 2}, {4}, {3})`) может потребоваться, чтобы элемент 1 оставался в `{1, 2}`.

Проблему можно решить клонированием списка при рекурсии. Также возможно действовать иначе: изменить список, но затем «откатить» изменения после завершения рекурсии.

Мы выбрали второй вариант. Так как по всему стеку рекурсивных вызовов проходит одна и та же ссылка на `first`, `second` и `prefix`, необходимо клонировать `prefix` непосредственно перед сохранением результата.

```
1  ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
2    ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();
3
4    if (node == null) {
5      result.add(new LinkedList<Integer>());
```

```
    return result;
  }

  LinkedList<Integer> prefix = new LinkedList<Integer>();
  prefix.add(node.data);

  /* Рекурсия по левому и правому поддереву. */
  ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
  ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

  /* Переплетение всех списков с левой и правой стороны. */
  for (LinkedList<Integer> left : leftSeq) {
    for (LinkedList<Integer> right : rightSeq) {
      ArrayList<LinkedList<Integer>> weaved =
        new ArrayList<LinkedList<Integer>>();
      weaveLists(left, right, weaved, prefix);
      result.addAll(weaved);
    }
  }
  return result;
}

/* Списки переплетаются всеми возможными способами. Алгоритм удаляет начало
 * из одного списка, проводит рекурсию, а затем проделывает то же
 * самое с другим списком. */
void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
      ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
  /* Один список пуст. Добавить остаток в [клонированный] prefix и сохранить
     результат. */
  if (first.size() == 0 || second.size() == 0) {
    LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
    result.addAll(first);
    result.addAll(second);
    results.add(result);
    return;
  }

  /* Рекурсия с началом first, добавленным в prefix. Удаление начала
   * модифицирует first, поэтому в дальнейшем его необходимо вернуть на место.
*/
  int headFirst = first.removeFirst();
  prefix.addLast(headFirst);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  first.addFirst(headFirst);

  /* Проделать то же с second, с модификацией и восстановлением списка.*/
  int headSecond = second.removeFirst();
  prefix.addLast(headSecond);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  second.addFirst(headSecond);
}
```

У некоторых людей эта задача вызывает трудности, потому что для ее решения необходимо спроектировать и реализовать два рекурсивных алгоритма. Они путаются

с тем, как алгоритмы должны взаимодействовать друг с другом, и пытаются мысленно объединить их.

Если вы принадлежите к числу таких людей, попробуйте действовать методом *доверия и концентрации*. Доверяйте правильности работы независимо реализованного метода и сконцентрируйтесь на том, что должен делать этот независимый метод.

Взгляните на метод `weaveLists`. Он выполняет конкретную операцию: строит все возможные переплетения двух списков и возвращает список переплетений. Существование `allSequences` к делу не относится. Сосредоточьтесь на задаче, которую должен выполнять метод `weaveLists`, и спроектируйте алгоритм.

Если вы реализуете `allSequences` (неважно, до или после `weaveLists`), доверьтесь методу `weaveLists` и считайте, что он работает правильно. Не отвлекайтесь на подробности работы метода `weaveLists` при реализации операции, которая от него совершенно не зависит. Сконцентрируйтесь на том, чем вы занимаетесь в настоящий момент.

Собственно, этот совет пригодится вам в любой ситуации, когда вы оказываетесь в тупике во время проработки общей логики программы. Постарайтесь четко понять, что должна делать конкретная функция («Так, эта функция должна возвращать список _____»). Убедитесь в том, что она действительно делает именно то, что требуется. Но когда вы не работаете над этой функцией, сконцентрируйтесь на текущих делах и доверьтесь тому, что другие функции правильно справляются со своим делом. Одновременно держать в голове реализации нескольких алгоритмов часто бывает слишком сложно.

4.10. T1 и T2 — два очень больших бинарных дерева, причем T1 значительно больше T2. Создайте алгоритм, проверяющий, является ли T2 поддеревом T1.

Дерево T2 считается поддеревом T1, если существует такой узел *n* в T1, что поддерево, «растущее» из *n*, идентично дереву T2. (Иначе говоря, если вырезать дерево в узле *n*, оно будет идентично T2.)

РЕШЕНИЕ

Задачи такого рода полезно сначала решить для небольшого количества данных. Это позволит сформировать общую концепцию решения.

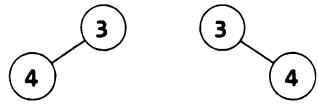
Упрощенная задача

В упрощенной задаче можно рассмотреть сравнение строковых представлений обходов дерева. Если `T2` является поддеревом `T1`, то обход `T2` будет подстрокой `T1`. Справедливо ли обратное? Если справедливо, то какой обход следует использовать — симметричный или префиксный?

Симметричный обход определенно не сработает. Возьмите хотя бы сценарий с бинарными деревьями поиска. При симметричном обходе бинарного дерева поиска значения всегда выводятся в отсортированном порядке. Таким образом, два бинарных дерева поиска с одинаковыми значениями всегда имеют одинаковые симметричные обходы, даже если их структура различается.

Как насчет префиксного обхода? Этот вариант чуть более перспективен. По крайней мере, в этом случае кое-что известно, например то, что первый элемент префиксного обхода является корневым узлом. За ним следуют левый и правый элементы.

К сожалению, деревья с разной структурой по-прежнему могут иметь одинаковый порядок префиксного обхода.

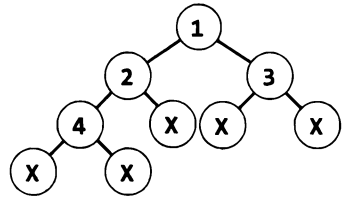


Впрочем, у этой проблемы есть простое решение. Можно представить NULL-узлы в строке префиксного обхода специальным символом, например X. (Предполагается, что бинарные деревья содержат только целые числа.) Левому дереву будет соответствовать строка обхода {3, 4, X}, а правому — строка {3, X, 4}.

Заметим, что при специальном представлении NULL-узлов префиксный обход дерева уникален. Иначе говоря, если два дерева имеют одинаковый порядок префиксного обхода, значит, они идентичны по значениям и структуре.

Чтобы убедиться в этом, рассмотрим построение дерева по его строке префиксного обхода (с обозначениями NULL-узлов), например 1, 2, 4, X, X, X, 3, X, X.

Сначала идет корень 1, за ним следует его левый узел 2. Левым узлом 2 должен быть узел 4. У 4 должно быть два NULL-узла (так как за ним следуют два X). Узел 4 завершен, поэтому мы возвращаемся к его родителю 2. Правым узлом 2 снова является X (NULL). Левое поддерево 1 завершено, переходим к правому дочернему узлу 1. Здесь помещается элемент 3 с двумя дочерними узлами NULL. Построение дерева завершено.



Весь процесс был детерминированным, как и для любого другого дерева. Префиксный обход всегда начинается с корня, и дальнейший путь полностью определяется обходом. Следовательно, два дерева идентичны, если они имеют одинаковый порядок префиксного обхода.

Теперь рассмотрим задачу поддерева. Если строка префиксного обхода T2 является подстрокой строки префиксного обхода T1, то корневой элемент T2 должен присутствовать в T1. Если выполнить префиксный обход из этого элемента в T1, мы будем следовать пути, идентичному обходу T2. Следовательно, T2 является поддеревом T1.

Реализация получается достаточно прямолинейной: нужно лишь построить и сравнить префиксные обходы.

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2    StringBuilder string1 = new StringBuilder();
3    StringBuilder string2 = new StringBuilder();
```

```
4
5    getOrderString(t1, string1);
6    getOrderString(t2, string2);
7
8    return string1.indexOf(string2.toString()) != -1;
9  }
10
11 void getOrderString(TreeNode node, StringBuilder sb) {
12   if (node == null) {
13     sb.append("X");              // Добавление индикатора null
14     return;
15   }
16   sb.append(node.data + " ");     // Добавление корня
17   getOrderString(node.left, sb);  // Добавление левого узла
18   getOrderString(node.right, sb); // Добавление правого узла
19 }
```

Это решение выполняется за время $O(n+m)$ и требует памяти $O(n+m)$, где n и m — количество узлов в `T1` и `T2` соответственно. При миллионах узлов такая пространственная сложность может оказаться неприемлемой.

Альтернативный подход

Альтернативное решение основано на поиске по большему дереву `T1`. Каждый раз, когда узел `T1` совпадает с корнем `T2`, вызывается метод `treeMatch`, который сравнит два поддерева и проверит их идентичность.

Анализ времени выполнения достаточно трудоемок. Было бы наивно заявить, что временная сложность составит $O(nm)$, где n — количество узлов в `T1`, а m — количество узлов в `T2`. Формально это выглядит правдоподобно, но на практике можно предоставить более точную границу.

Мы не вызываем метод `matchTree` для каждого узла `T2`. Мы вызываем его `k` раз, где `k` — количество вхождений корня `T2` в `T1`, поэтому время выполнения ближе к $O(n + km)$.

На самом деле даже эта оценка несколько завышена. Даже если корень совпал, мы выходим из `matchTree`, как только обнаруживается различие между `T1` и `T2`. Поэтому, скорее всего, нам не приходится просматривать все m узлов при каждом вызове `treeMatch`.

Приведенный далее код реализует этот алгоритм:

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2    if (t2 == null) return true; // Пустое дерево всегда является поддеревом
3    return subTree(t1, t2);
4  }
5
6  boolean subTree(TreeNode r1, TreeNode r2) {
7    if (r1 == null) {
8      return false; // Большое дерево пустое, а поддерево так и не найдено.
9    } else if (r1.data == r2.data && matchTree(r1, r2)) {
10     return true;
11   }
12   return subTree(r1.left, r2) || subTree(r1.right, r2);
13 }
```

```
14
15  boolean matchTree(TreeNode r1, TreeNode r2) {
16    if (r1 == null && r2 == null) {
17      return true; // В поддереве не осталось узлов
18    } else if (r1 == null || r2 == null) {
19      return false; // Ровно одно дерево пусто, поэтому совпадения нет
20    } else if (r1.data != r2.data) {
21      return false;  // Данные не совпадают
22    } else {
23      return matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right);
24    }
25  }
```

Когда лучше более простое решение? Будет ли лучше альтернативное решение? Это тема для разговора с интервьюером. Вот несколько советов по этому поводу.

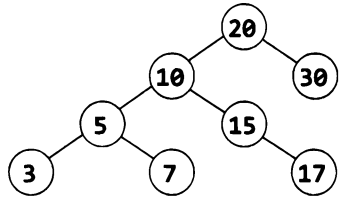
1. Простое решение занимает $O(n + m)$ памяти, а альтернативное — $O(\log(n) + \log(m))$. Не забывайте, что затраты памяти могут оказаться очень важным фактором, когда речь заходит о масштабируемости.
2. Простое решение потребует $O(n + m)$ времени, а альтернативное — $O(nm)$ в худшем случае. Впрочем, сложность худшего случая обманчива; следует смотреть глубже.
3. Как мы уже выяснили ранее, для времени выполнения можно дать более строгую оценку $O(n + km)$, где `k` — количество вхождений корня `T2` в `T1`. Предположим, что данные узлов в `T1` и `T2` — случайные величины в диапазоне от 0 до `p`. Тогда значение `k` равно приблизительно `n/p`. Почему? Каждый из `n` узлов в `T1` может быть равен корню `T2.root` с вероятностью `1/p`. Пусть `p = 1000` `n = 1000000` и `m = 100`. Тогда нам понадобится выполнить около 1 100 000 проверок узлов (`1100000 = 1000000 + 100*1000000/1000`).
4. Более сложные математические расчеты позволят дополнительно уточнить эту оценку. Ранее мы предположили, что при вызове `matchTree` нам придется обойти m узлов `T2`, хотя, вероятно, мы скорее найдем различия в начале дерева, чем в конце, а следовательно, раньше выйдем.

В целом альтернативное решение оказывается более эффективным с точки зрения пространства и, скорее всего, также и с точки зрения времени выполнения. Впрочем, все зависит от предположений и от того, насколько важно сократить время выполнения среднего случая за счет худшего случая. Будет очень хорошо, если вы упомянете об этом в разговоре с интервьюером.

4.11. Вы пишете с нуля класс бинарного дерева поиска, который помимо методов вставки, поиска и удаления содержит метод `getRandomNode()` для получения случайного узла дерева. Вероятность выбора всех узлов должна быть одинаковой. Разработайте и реализуйте алгоритм `getRandomNode`; объясните, как вы реализуете остальные методы.

РЕШЕНИЕ

Возьмем пример для наглядности.



Мы исследуем несколько решений, пока не найдем работоспособное и оптимальное.

Прежде всего следует заметить, что задача сформулирована весьма интересно. Вам не предлагается спроектировать алгоритм, возвращающий случайный узел из бинарного дерева, — сказано, что этот класс строится с нуля. Такая формулировка не случайна: вероятно, нам понадобится доступ к внутренностям структуры данных.

Решение 1 (медленное и работоспособное)

Очевидное решение — скопировать все узлы в массив и вернуть случайный элемент массива. Такое решение выполняется за время $O(N)$ с затратами памяти $O(N)$, где N — количество узлов в дереве.

Вероятно, интервьюер хочет получить от нас что-то более эффективное, потому что это решение слишком прямолинейно (а также оставляет вопрос, почему речь идет именно о бинарном дереве, — эта информация нами не использовалась).

При разработке решения следует учесть, что нам, вероятно, нужно что-то знать о внутреннем строении дерева. В противном случае в формулировке задачи не было бы сказано, что класс дерева создается с нуля.

Решение 2 (медленное и работоспособное)

Возвращаясь к исходному решению с копированием узлов в массиве, можно исследовать решение, в котором постоянно поддерживается массив со всеми узлами дерева. Проблема в том, что нам придется удалять элементы из массива при удалении их из дерева, а эта операция выполняется за время $O(N)$.

Решение 3 (медленное и работоспособное)

Всем узлам можно назначить индексы от 1 до N в порядке бинарного дерева поиска (то есть в соответствии с симметричным обходом). При вызове `getRandomNode` генерируется случайный индекс от 1 до N. Если правильно воспользоваться значением индекса, для его поиска можно воспользоваться бинарным деревом поиска.

Однако при этом возникает та же проблема, что и в предыдущих решениях. Вставка или удаление узла могут потребовать обновления всех индексов, а это может потребовать времени $O(N)$.

Решение 4 (быстрое и неработоспособное)

А если глубина дерева известна? (Так как мы строим класс сами, можно позаботиться о том, чтобы она всегда была известна. Организовать отслеживание этого значения нетрудно.)

Можно выбрать случайную глубину и затем выполнить случайный обход влево/вправо, пока не будет достигнута нужная глубина. Однако такой подход не гарантирует, что все узлы будут выбираться с равной вероятностью.

Во-первых, дерево не обязательно имеет равное количество узлов на всех уровнях. Это означает, что узлы на уровнях с меньшим количеством узлов будут выбираться с меньшей вероятностью, чем узлы на уровнях с большим количеством узлов.

Во-вторых, случайный путь может завершиться до того, как мы доберемся до нужного уровня. И что делать в этом случае? Можно просто вернуть последний найденный узел, но это будет означать неравенство вероятностей на уровнях.

Решение 5 (быстрое и неработоспособное)

Еще один упрощенный подход: выполняем случайный обход вниз по дереву. В каждом узле:

- с вероятностью 1/3 вернуть текущий узел.
- с вероятностью 1/3 пройти по левой ветви.
- с вероятностью 1/3 пройти по правой ветви.

Это решение, как и некоторые другие, не обеспечивает равномерного распределения вероятностей между узлами. Корень выбирается с вероятностью 1/3 — с такой же, как у *всех* узлов левого поддерева в сумме.

Решение 6 (быстрое и работоспособное)

Вместо того чтобы продолжать плодить новые решения, попробуем исправить некоторые недостатки предыдущих решений. Для этого необходимо проанализировать — притом глубоко! — главную причину проблем с предыдущими решениями.

Рассмотрим решение 5. Оно не работает из-за того, что вероятности неравномерно распределяются между решениями. Можно ли исправить недостаток, не изменяя базовый алгоритм?

Начнем с корня. С какой вероятностью должен возвращаться корень? Дерево содержит N узлов, поэтому корневой узел должен возвращаться с вероятностью $1/N$ — как, собственно, и все остальные узлы.

Проблема с корнем решена. Как насчет остального? С какой вероятностью следует выбирать между движением по левому и правому поддереву? Не 50/50. Даже в сбалансированном дереве количество узлов на двух сторонах может быть разным. Если в левом поддереве больше узлов, чем в правом, то левое поддерево должно выбираться с большей вероятностью.

Вероятность выбора в левом поддереве должна быть равна сумме отдельных вероятностей. Так как каждый узел имеет вероятность $1/N$, вероятность выбора в левом поддереве должна быть равна `РАЗМЕР_ЛЕВОГО_ПОДДЕРЕВА * 1/N`. Аналогичным образом

вероятность выбора в правом поддереве должна быть равна РАЗМЕР_ПРАВОГО_ПОДДЕРЕВА * 1/N.

Это означает, что каждый узел должен знать размер своего левого и правого поддерева. К счастью, интервьюер сообщил, что мы строим класс дерева с нуля, что позволяет легко отслеживать информацию о размере при выполнении вставки и удаления. Для этого в каждый узел добавляется новая переменная со значением размера, которая увеличивается при вставке и уменьшается при удалении.

```
1  class TreeNode {
2    private int data;
3    public TreeNode left;
4    public TreeNode right;
5    private int size = 0;
6
7    public TreeNode(int d) {
8      data = d;
9      size = 1;
10   }
11
12   public TreeNode getRandomNode() {
13     int leftSize = left == null ? 0 : left.size();
14     Random random = new Random();
15     int index = random.nextInt(size);
16     if (index < leftSize) {
17       return left.getRandomNode();
18     } else if (index == leftSize) {
19       return this;
20     } else {
21       return right.getRandomNode();
22    }
23   }
24
25   public void insertInOrder(int d) {
26     if (d <= data) {
27       if (left == null) {
28         left = new TreeNode(d);
29       } else {
30         left.insertInOrder(d);
31     }
32     } else {
33       if (right == null) {
34         right = new TreeNode(d);
35       } else {
36         right.insertInOrder(d);
37     }
38    }
39     size++;
40   }
41
42   public int size() { return size; }
43   public int data() { return data; }
44
45   public TreeNode find(int d) {
46     if (d == data) {
47       return this;
```

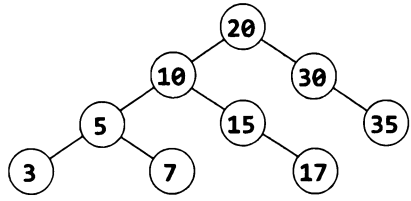
```
48      } else if (d <= data) {
49        return left != null ? left.find(d) : null;
50      } else if (d > data) {
51        return right != null ? right.find(d) : null;
52     }
53      return null;
54    }
55  }
```

В сбалансированном дереве этот алгоритм выполняется за время $O(\log N)$, где N — количество узлов.

Решение 7 (быстрое и работоспособное)

Вызовы генератора случайных чисел могут обходиться дорого. При желании их количество можно существенно сократить.

Представьте, что мы вызвали `getRandomNode` для изображенного ниже дерева и выбрали левый путь.



Мы пошли налево, потому что было выбрано число от 0 до 5 (включительно). Затем мы снова выбираем случайное число от 0 до 5. Зачем? С таким же успехом можно воспользоваться и первым.

А если бы мы вместо этого пошли направо? Значит, было получено число от 7 до 8 (включительно), а после этого нужно число от 0 до 1 (включительно). Проблема решается легко: достаточно вычесть `РАЗМЕР_ЛЕВОГО_ПОДДЕРЕВА + 1`.

Происходящее можно объяснить иначе: исходное случайное число указывает, какой узел (`i`) следует вернуть, после чего мы ищем `i`-й узел в порядке симметричного обхода. Вычитание `РАЗМЕР_ЛЕВОГО_ПОДДЕРЕВА + 1` из `i` отражает тот факт, что при выборе правого пути из симметричного обхода исключаются `РАЗМЕР_ЛЕВОГО_ПОДДЕРЕВА + 1` узлов.

```
1  class Tree {
2    TreeNode root = null;
3
4    public int size() { return root == null ? 0 : root.size(); }
5
6    public TreeNode getRandomNode() {
7      if (root == null) return null;
8
9      Random random = new Random();
10      int i = random.nextInt(size());
11      return root.getIthNode(i);
12    }
13
14    public void insertInOrder(int value) {
```

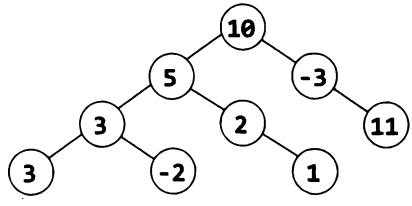
```
15      if (root == null) {
16        root = new TreeNode(value);
17      } else {
18        root.insertInOrder(value);
19     }
20    }
21  }
22
23  class TreeNode {
24    /* Конструктор и переменные остались теми же. */
25
26    public TreeNode getIthNode(int i) {
27      int leftSize = left == null ? 0 : left.size();
28      if (i < leftSize) {
29        return left.getIthNode(i);
30      } else if (i == leftSize) {
31        return this;
32      } else {
33        /* Пропускаются leftSize + 1 узлов, вычитаем их. */
34        return right.getIthNode(i - (leftSize + 1));
35     }
36    }
37
38    public void insertInOrder(int d) { /* Без изменений */ }
39    public int size() { return size; }
40    public TreeNode find(int d) { /* Без изменений */ }
41  }
```

Этот алгоритм, как и предыдущий, выполняется в сбалансированном дереве за время $O(\log N)$. Время выполнения также можно описать в виде $O(D)$, где D — максимальная глубина дерева. Обратите внимание: оценка $O(D)$ точно описывает время выполнения независимо от того, сбалансированно дерево или нет.

4.12. Дано бинарное дерево, в котором каждый узел содержит целое число (положительное или отрицательное). Разработайте алгоритм для подсчета всех путей, сумма значений которых соответствует заданной величине. Обратите внимание, что путь не обязан начинаться или заканчиваться в корневом или листовом узле, но он должен идти вниз (переходя только от родительских узлов к дочерним).

РЕШЕНИЕ

Давайте выберем потенциальную сумму (скажем, 8) и нарисуем бинарное дерево. В следующем примере в дерево намеренно было включено несколько путей с такой суммой.



Попробуем действовать методом «грубой силы».

Решение 1. Метод «грубой силы»

В этом случае просто рассматриваются все возможные пути. Для этого мы переходим к каждому узлу и в каждом узле рекурсивно опробуем все пути, ведущие вниз, подсчитывая сумму в процессе перебора. При обнаружении целевой суммы увеличивается счетчик путей.

```
1  int countPathsWithSum(TreeNode root, int targetSum) {
2    if (root == null) return 0;
3
4    /* Подсчет путей с заданной суммой, начиная с корня. */
5    int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);
6
7    /* Проверка узлов слева и справа. */
8    int pathsOnLeft = countPathsWithSum(root.left, targetSum);
9    int pathsOnRight = countPathsWithSum(root.right, targetSum);
10
11   return pathsFromRoot + pathsOnLeft + pathsOnRight;
12 }
13
14 /* Возвращает количество путей с заданной суммой от данного узла. */
15 int countPathsWithSumFromNode(TreeNode node, int targetSum, int currentSum) {
16   if (node == null) return 0;
17
18   currentSum += node.data;
19
20   int totalPaths = 0;
21   if (currentSum == targetSum) { // Найден путь от корня
22     totalPaths++;
23   }
24
25   totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
26   totalPaths += countPathsWithSumFromNode(node.right, targetSum, currentSum);
27   return totalPaths;
28 }
```

Какова временная сложность этого алгоритма?

Узел на глубине d будет «опробован» (через `countPathsWithSumFromNode`) d узлами, находящимися над ним.

В сбалансированном бинарном дереве поиска d не превышает приблизительно $\log N$. Следовательно, мы знаем, что с N узлами в дереве метод `countPathsWithSumFromNode` будет вызван $O(N \log N)$ раз. Время выполнения составляет $O(N \log N)$.

Также к этому результату можно прийти с другой стороны. Из корневого узла обход осуществляется по всем $N - 1$ узлам, находящимся под ним (через `countPathsWithSumFromNode`). На втором уровне (с двумя узлами) обходятся $N - 3$ узла. На третьем уровне (четыре узла плюс три над ними) обход осуществляется по $N - 7$ узлам.

По этому образцу общий объем работы составит приблизительно:

$$(N - 1) + (N - 3) + (N - 7) + (N - 15) + (N - 31) + \ldots + (N - N)$$

Чтобы упростить эту запись, заметим, что в левой части каждого слагаемого всегда стоит N, а правая часть на 1 меньше степени 2. Количество слагаемых равно глубине дерева, то есть $O(\log N)$. Что касается правой части, уменьшение

степени 2 на 1 можно проигнорировать. Таким образом, фактически мы имеем следующее:

$O(N$ * [*количество слагаемых*] – [сумма степеней двойки от 1 до N])

$O(N \log N - N)$

$O(N \log N)$

Если значение суммы степеней 2 от 1 до N для вас не очевидно, подумайте, как выглядят степени 2 в двоичной записи:

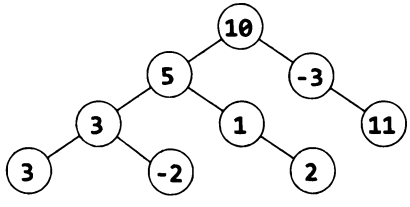
```
  0001
+ 0010
+ 0100
+ 1000
= 1111
```

Следовательно, в сбалансированном дереве время выполнения составит $O(N \log N)$.

В несбалансированном дереве оно может быть намного хуже. Рассмотрим дерево, которое представляет собой прямую линию, направленную вниз. В корне обходятся $N - 1$ узлов, на следующем уровне (с одним узлом) — $N - 2$ узла, на третьем уровне — $N - 3$ узла и т. д. В результате получаем сумму чисел от 1 до N, что соответствует $O(N^2)$.

Решение 2. Оптимизация

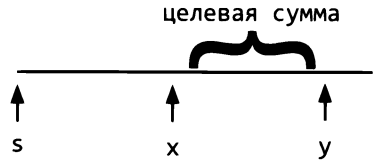
Анализируя приведенное решение, можно понять, что часть работы повторяется. Скажем, путь вида 10 -> 5 -> 3 -> -2 (или отдельные его части) обходится повторно. Сначала это происходит, когда мы начинаем с узла 10, затем при переходе к узлу 5 (проверяем 5, потом 3 и -2), затем при переходе к узлу 3 и, наконец, при переходе к узлу -2. В идеале хотелось бы использовать результаты, уже полученные ранее.



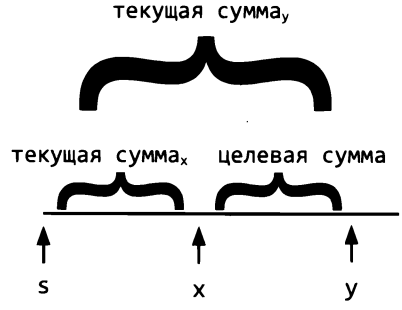
Выделим некоторый путь и будем рассматривать его как массив. Возьмем (гипотетический расширенный) путь вида

10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2

По сути, вопрос заключается в следующем: сколько непрерывных подпоследовательностей в этом массиве дают целевую сумму (например, 8)? Иначе говоря, для каждого `y` мы пытаемся найти значения `x` на следующей схеме (или, точнее, количество значений `x`).



Если каждому значению известна его текущая сумма (то есть сумма значений от `s` до него), тогда результат находится относительно просто. Достаточно воспользоваться простой формулой: $текущаяСумма_x = текущаяСумма_y$ - `целевая сумма`. Затем мы ищем значения `x`, для которых эта формула истинна.



Так как нас интересует только количество путей, можно воспользоваться хеш-таблицей. В процессе перебора массива построим хеш-таблицу, связывающую текущую сумму с количеством ее обнаружений. Затем для каждого `y` в хеш-таблице ищется $текущаяСумма_y$ - `целеваяСумма`. Значение в хеш-таблице сообщит количество путей с суммой `целеваяСумма`, заканчивающихся в `y`.

Пример:

```
индекс  :   0    1    2    3     4     5    6    7    8
значение:  10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
сумма   :  10   15   16   18    17    16   23   24   26
```

Значение $текущаяСумма_7$ равно 24. Если `целеваяСумма` равна 8, в хеш-таблице ищется значение 16. Поиск дает два таких значения (с индексами 2 и 5.) Как видно из приведенного примера, индексы с 3 по 7 и индексы с 6 по 7 имеют сумму 8.

Разобравшись с алгоритмом для массива, посмотрим, как он работает с деревом.

Мы будем действовать аналогично: для обхода дерева будет использоваться поиск в глубину. При посещении каждого узла:

1. Отслеживать его текущую сумму. Значение передается в параметре и немедленно увеличивается на `node.value`.
2. Найти в хеш-таблице `текущаяСумма-целеваяСумма`. Присвоить `totalPaths` найденное значение.
3. Если `текущаяСумма = целеваяСумма`, значит, существует еще один путь, начинающийся с корня. Увеличить значение `totalPaths`.

4. Добавить текущую сумму в хеш-таблицу (увеличить значение, если оно там уже присутствует).
5. Выполнить рекурсию по левому и правому пути, подсчитывая количество путей целевой суммой.
6. После завершения рекурсии уменьшить значение `текущаяСумма` в хеш-таблице. Фактически это возврат после проделанной работы; изменения в хеш-таблице отменяются, чтобы они не использовались другими узлами.

Несмотря на сложность разработки этого алгоритма, его программная реализация относительно проста.

```
int countPathsWithSum(TreeNode root, int targetSum) {
  return countPathsWithSum(root, targetSum, 0, new HashMap<Integer, Integer>());
}

int countPathsWithSum(TreeNode node, int targetSum, int runningSum,
                      HashMap<Integer, Integer> pathCount) {
  if (node == null) return 0; // Базовый случай

  /* Подсчет путей с суммой, заканчивающихся на текущем узле. */
  runningSum += node.data;
  int sum = runningSum - targetSum;
  int totalPaths = pathCount.getOrDefault(sum, 0);

  /* Если runningSum == targetSum, один дополнительный путь начинается
   * от корня. Добавить этот путь. */
  if (runningSum == targetSum) {
    totalPaths++;
  }

  /* Увеличение pathCount, рекурсия и уменьшение pathCount. */
  incrementHashTable(pathCount, runningSum, 1); // Увеличение pathCount
  totalPaths += countPathsWithSum(node.left, targetSum, runningSum, pathCount);
  totalPaths += countPathsWithSum(node.right, targetSum, runningSum, pathCount);
  incrementHashTable(pathCount, runningSum, -1); // Уменьшение pathCount

  return totalPaths;
}

void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int delta) {
  if (newCount == 0) { // Удалить для экономии памяти
    hashTable.remove(key);
  } else {
    hashTable.put(key, newCount);
  }
}
```

Время выполнения алгоритма составляет $O(N)$, где N — количество узлов в дереве. Оценка времени выполнения основана на том, что каждый узел посещается всего один раз и каждый раз выполняется работа $O(1)$. В сбалансированном дереве пространственная сложность равна $O(\log N)$ из-за хеш-таблицы. В несбалансированном дереве пространственная сложность может вырасти до $O(n)$.

5

Операции с битами

5.1. Даны два 32-разрядных числа N и M и две позиции битов i и j. Напишите метод для вставки M в N так, чтобы число M занимало позицию с бита j по бит i. Предполагается, что j и i имеют такие значения, что число M гарантированно поместится в этот промежуток. Скажем, для M = 10011 можно считать, что j и i разделены как минимум 5 битами. Комбинация вида j = 3 и i = 2 невозможна, так как число M не поместится между битом 3 и битом 2.

Пример:

Ввод: N = 10000000000, M = 10011, i = 2, j = 6

Вывод: N = 10001001100

РЕШЕНИЕ

Решение этой задачи можно разделить на три основных шага:

1. Сбросить в N биты с j по i.
2. Сдвинуть M так, чтобы оно оказалось под позициями j до i.
3. Соединить M и N.

Самая сложная часть — шаг 1. Как сбросить биты в N? Это можно сделать с помощью маски. Во всех позициях маски содержатся единицы, кроме интервала с j по i (в этих позициях должны стоять нули). Мы сначала создадим левую половину маски, а затем правую.

```
1  int updateBits(int n, int m, int i, int j) {
2    /* Создание маски для сброса битов с i по j в n. ПРИМЕР: i = 2, j = 4.
3     * Результат равен 11100011. Для простоты ограничимся 8 битами. */
4    int allOnes = ~0; // Последовательность из единиц
5
6    // Единицы до позиции j, потом нули. left = 11100000
7    int left = allOnes << (j + 1);
8
9    // Единицы после позиции i. right = 00000011
10    int right = ((1 << i) - 1);
11
12    // Все единицы, нули только в позициях от i до j. mask = 11100011
13    int mask = left | right;
14
15    /* Сбросить биты с j по i, затем поместить m. */
16    int n_cleared = n & mask; // Сброс битов с j по i.
17    int m_shifted = m << i; // Переместить m в правильную позицию.
18
19    return n_cleared | m_shifted; // Остается применить операцию OR
20  }
```

В этой задаче (и в других задачах, использующих операции с битами) код необходимо тщательно протестировать из-за повышенного риска ошибок смещения на 1.

5.2. Дано вещественное число в интервале между 0 и 1 (например, 0,72), которое передается в формате double. Выведите его двоичное представление. Если для точного двоичного представления числа не хватает 32 разрядов, выведите сообщение об ошибке.

РЕШЕНИЕ

Чтобы исключить неоднозначность, мы будем использовать нижние индексы x_2 и x_{10} для обозначения системы счисления (двоичная или десятичная).

Как выглядит нецелое число в двоичном представлении? По аналогии с десятичным числом двоичное число 0.101_2 можно записать как

$$0.101_2 = 1 * (1/2^1) + 0 * (1/2^2) + 1 * (1/2^3)$$

Чтобы вывести дробную часть, мы можем умножить число на 2 и сравнить 2n с 1. По сути, происходит «сдвиг» дробной суммы:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * (1/2^0) + 0 * (1/2^1) + 1 * (1/2^2) \\ &= 1.01_2 \end{aligned}$$

Если r >= 1, то сразу понятно, что после десятичной точки в n стоит 1. Многократно выполняя эту операцию, можно проверить каждую цифру.

```
1  String printBinary(double num) {
2    if (num >= 1 || num <= 0) {
3      return "ERROR";
4    }
5
6    StringBuilder binary = new StringBuilder();
7    binary.append(".");
8    while (num > 0) {
9      /* Ограничение длины: 32 символа */
10     if (binary.length() >= 32) {
11       return "ERROR";
12     }
13
14     double r = num * 2;
15     if (r >= 1) {
16       binary.append(1);
17       num = r - 1;
18     } else {
19       binary.append(0);
20       num = r;
21     }
22   }
23   return binary.toString();
24 }
```

Также можно действовать иначе: вместо умножения числа на 2 и сравнения с 1 сравнить число с 0,5, потом с 0,25 и т. д., как показано ниже:

```
String printBinary2(double num) {
  if (num >= 1 || num <= 0) {
    return "ERROR";
  }

  StringBuilder binary = new StringBuilder();
  double frac = 0.5;
  binary.append(".");
  while (num > 0) {
    /* Ограничение длины: 32 символа */
    if (binary.length() > 32) {
      return "ERROR";
    }
    if (num >= frac) {
      binary.append(1);
      num -= frac;
    } else {
      binary.append(0);
    }
    frac /= 2;
  }
  return binary.toString();
}
```

Оба варианта одинаково хороши. Используйте тот, который вам больше нравится.

Что бы вы ни выбрали, обязательно предусмотрите все возможные тестовые сценарии и обсудите их с интервьюером.

5.3. Имеется целое число, в котором можно изменить ровно один бит из 0 в 1. Напишите код для определения длины самой длинной последовательности единиц, которая может быть при этом получена.

Пример:

Ввод: `1775 (или: 11011101111)`

Вывод: `8`

РЕШЕНИЕ

Любое целое число может рассматриваться как последовательность чередующихся 1 и 0. Если последовательность 0 имеет длину 1, появляется потенциальная возможность слияния соседних последовательностей 1.

Метод «грубой силы»

Целое число можно преобразовать в массив, отражающий длины последовательностей 0 и 1. Например, последовательности 11011101111 будет соответствовать следующий массив (справа налево) $[0_0, 4_1, 1_0, 3_1, 1_0, 2_1, 21_0]$. Нижний индекс указывает, соответствует ли число последовательности 0 или 1, но для реального решения эта информация не нужна. Последовательность всегда строго чередуется и всегда начинается с 0.

При наличии такого массива мы просто перебираем его элементы. Для каждой последовательности 0 рассматривается возможность слияния соседних последовательностей 1, если последовательность 0 имеет длину 1.

```
int longestSequence(int n) {
  if (n == -1) return Integer.BYTES * 8;
  ArrayList<Integer> sequences = getAlternatingSequences(n);
  return findLongestSequence(sequences);
}

/* Получение списка размеров последовательностей. Первая
 * последовательность всегда состоит из 0 (и может иметь нулевую длину),
 * а далее следуют длины чередующихся последовательностей 1 и 0.*/
ArrayList<Integer> getAlternatingSequences(int n) {
  ArrayList<Integer> sequences = new ArrayList<Integer>();

  int searchingFor = 0;
  int counter = 0;

  for (int i = 0; i < Integer.BYTES * 8; i++) {
    if ((n & 1) != searchingFor) {
      sequences.add(counter);
      searchingFor = n & 1; // Переключение 1 в 0 или 0 в 1
      counter = 0;
   }
    counter++;
    n >>>= 1;
  }
  sequences.add(counter);

  return sequences;
}

/* Для заданных длин чередующихся последовательностей 0 и 1
 * найти самую длинную, которую можно построить. */
int findLongestSequence(ArrayList<Integer> seq) {
  int maxSeq = 1;

  for (int i = 0; i < seq.size(); i += 2) {
    int zerosSeq = seq.get(i);
    int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
    int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;

    int thisSeq = 0;
    if (zerosSeq == 1) { // Возможно слияние
      thisSeq = onesSeqLeft + 1 + onesSeqRight;
    } if (zerosSeq > 1) { // Добавить нуль с любой из сторон
      thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
    } else if (zerosSeq == 0) { // Выбрать с любой стороны
      thisSeq = Math.max(onesSeqRight, onesSeqLeft);
   }
    maxSeq = Math.max(thisSeq, maxSeq);
  }

  return maxSeq;
}
```

Решение получается довольно эффективным. Оно требует затрат времени $O(b)$ и затрат памяти $O(b)$, где b — длина последовательности.

Будьте внимательны с выражением времени выполнения. Например, если сказать, что время выполнения равно $O(n)$, что есть n? Было бы неправильно утверждать, что алгоритм имеет сложность O(*целое значение*): он имеет сложность O(*количество битов*). По этой причине, если использование n создает потенциальную неоднозначность, лучше не использовать эту запись, чтобы не путаться самому и не путать интервьюера. Выберите другое имя переменной (мы выбрали b, от слова «bits»).

Возможно ли получить лучший результат? В данном случае читать последовательность приходится всегда, поэтому оптимизировать решение по времени не удастся. С другой стороны, можно сократить затраты памяти.

Оптимальный алгоритм

Чтобы сократить затраты памяти, обратите внимание на то, что нам не нужно постоянно хранить длину каждой последовательности. Она должна храниться столько времени, сколько необходимо для сравнения каждой последовательности 1 с предшествующей ей последовательностью 1.

Следовательно, мы можем ограничиться отслеживанием длин текущей и предыдущей последовательности 1. При обнаружении нулевого бита происходит обновление предыдущей длины `previousLength`:

- Если следующий бит содержит 1, `previousLength` присваивается `currentLength`.
- Если следующий бит содержит 0, выполнить слияние последовательностей невозможно, поэтому `previousLength` присваивается 0.

Значение `maxLength` обновляется в процессе перебора.

```
1  int flipBit(int a) {
2    /* Если a содержит только 1, это уже самая длинная последовательность. */
3    if (~a == 0) return Integer.BYTES * 8;
4
5    int currentLength = 0;
6    int previousLength = 0;
7    int maxLength = 1; // Всегда может быть последовательность, содержащая
                        не менее одной 1
8    while (a != 0) {
9      if ((a & 1) == 1) { // Текущий бит равен 1
10       currentLength++;
11     } else if ((a & 1) == 0) { // Текущий бит равен 0
12       /* Присвоить 0 (следующий бит равен 0) или currentLength
                       (следующий бит равен 11). */
13       previousLength = (a & 2) == 0 ? 0 : currentLength;
14       currentLength = 0;
15     }
16     maxLength = Math.max(previousLength + currentLength + 1, maxLength);
17     a >>>= 1;
18   }
19   return maxLength;
20 }
```

Время выполнения этого алгоритма также равно $O(b)$, но объем дополнительной памяти сокращается до $O(1)$.

5.4. Для заданного положительного числа выведите ближайшие наименьшее и наибольшее числа, которые имеют такое же количество единичных битов в двоичном представлении.

РЕШЕНИЕ

Существует несколько способов решения этой задачи, в том числе метод «грубой силы», манипуляции с битами, нетривиальные арифметические операции и т. д. Обратите внимание: арифметический подход основан на манипуляциях с битами, поэтому арифметический метод будет рассматриваться в последнюю очередь.

Не запутайтесь с терминологией в этой задаче. Метод `getNext` будет возвращать большее число, а `getPrev` — меньшее.

Метод «грубой силы»

Проще всего действовать методом «грубой силы»: подсчитываем количество единиц в `n`, а затем постепенно увеличиваем (или уменьшаем) `n`, пока не найдем число с таким же количеством единиц. Метод прост, но неинтересен. Можем ли мы решить задачу оптимальнее? Да!

Начнем с кода метода `getNext`, а затем перейдем к `getPrev`.

Поразрядная обработка для getNext

Подумаем, каким *должно быть* следующее число? Допустим, имеется число 13 948; его двоичное представление имеет вид:

1	1	0	1	1	0	0	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Нам нужно получить ближайшее большее число и сохранить исходное количество единиц.

Допустим, в числе `n` с позициями битов `i` и `j` мы инвертируем значение бита `i` из 1 в 0, а бита `j` — из 0 в 1. Если `i > j`, то число `n` уменьшится. Если `i < j`, то число `n` увеличится.

Из этого можно сделать следующие выводы:

1. Если мы инвертируем значение одного из битов из 0 в 1, то другой бит переходит из 1 в 0.
2. Число окажется больше в том, и только в том случае, если бит 0->1 находился левее бита 1->0.
3. Результирующее число должно быть больше, но не чрезмерно. То есть нам необходимо инвертировать самый правый 0, у которого справа находится 1.

Иначе говоря, нужно инвертировать самый правый нуль, который не является завершающим. В нашем примере завершающие нули находятся в позициях 0 и 1, а самый правый нуль, который не является завершающим, — в позиции 7. Обозначим эту позицию `p`.

Шаг 1. Инвертирование правого нуля, который не является завершающим

1	1	0	1	1	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

С этим изменением мы увеличили n. С другой стороны, в числе появилась одна лишняя единица и не хватает одного нуля. Число нужно уменьшить, насколько это возможно, не забывая об этом обстоятельстве.

Чтобы уменьшить число, мы переставим все биты справа от p так, чтобы 0 располагались слева, а 1 — справа. При этом одну из единиц нужно будет заменить 0.

Относительно простое решение — подсчитать количество единиц справа от p, сбросить все биты от 0 до p, а затем вернуть `c1 - 1` единиц (пусть `c1` — число единиц справа от p, а `c0` — количество нулей справа от p).

Рассмотрим работу этого алгоритма на примере:

Шаг 2. Сброс битов, находящихся справа от p (c0 = 2, c1 = 5, p = 7)

1	1	0	1	1	0	1	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Для сброса битов нужно создать маску, представляющую собой последовательность единиц, за которой следуют p нулей:

```
a = 1 << pos; // все нули, за исключением 1 на позиции p.
b = a - 1;    // все нули, за которыми следуют p единиц.
mask = ~b;    // все единицы, за которыми следуют p нулей.
n = n & mask; // сброс p самых правых битов.
```

В сокращенной записи:

```
n &= ~((1 << pos) - 1).
```

Шаг 3. Добавление c1 – 1 единиц

1	1	0	1	1	0	1	0	0	0	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Чтобы вставить справа `c1 - 1` единиц, выполните следующие действия:

```
a = 1 << (c1 - 1); // нули с 1 в позиции c1
b = a - 1;         // нули с 1 в позициях с 0 по c1 - 1
n = n | b;         // вставка 1 в позициях с 0 по c1 - 1
```

В сокращенной записи:

```
n |= (1 << (c1 - 1)) - 1;
```

Таким образом, мы получили наименьшее число, превышающее n, и с таким же количеством единиц.

Код `getNext` выглядит так:

```
1  int getNext(int n) {
2    /* Вычисление c0 и c1 */
```

```
3    int c = n;
4    int c0 = 0;
5    int c1 = 0;
6    while (((c & 1) == 0) && (c != 0)) {
7      c0++;
8      c >>= 1;
9    }
10
11    while ((c & 1) == 1) {
12      c1++;
13      c >>= 1;
14    }
15
16    /* Ошибка: если n == 11..1100...00, то большего числа с таким же
17     * количеством единиц не существует. */
18    if (c0 + c1 == 31 || c0 + c1 == 0) {
19      return -1;
20    }
21
22    int p = c0 + c1; // Позиция крайнего правого незавершающего нуля
23
24    n |= (1 << p); // Переключение крайнего правого незавершающего нуля
25    n &= ~((1 << p) - 1); // Сброс всех битов справа от p
26    n |= (1 << (c1 - 1)) - 1; // Вставка (c1 - 1) единиц справа.
27    return n;
28  }
```

Манипуляция с битами в getPrev

В реализации `getPrev` можно использовать аналогичный подход:

1. Вычислить `c0` и `c1`. `c1` — это число завершающих нулей, а `c0` — размер блока нулей, расположенных слева от завершающих нулей.
2. Заменить все незавершающие единицы нулями. Это будут позиции `p = c1 + c0`.
3. Сбросить все биты, находящиеся справа от бита `p`.
4. Установить в единицу `c1 + 1` бит справа от позиции `p`.

Обратите внимание, что шаг 2 устанавливает бит `p` в 0, а шаг 3 устанавливает биты от 0 до `p-1` в 0. Эти два шага можно объединить.

Рассмотрим работу этого алгоритма на примере.

Шаг 1. Исходное число. p = 7, c1 = 2, c0 = 5.

1	0	0	1	1	1	1	0	0	0	0	0	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Шаги 2 и 3. Сброс битов с 0 до p

1	0	0	1	1	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Можно сделать так:

```
int a = ~0;           // последовательность единиц
int b = a << (p + 1); // последовательность нулей, за которыми следуют P + 1 единиц
n &= b;               // Очищаем биты с 0 по p.
```

Шаг 4. Вставка (c1 + 1) единиц справа от p

1	0	0	1	1	1	0	1	1	1	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Поскольку `p = c1 + c0`, `(c1 + 1)` единиц будут сопровождаться `(c0 – 1)` нулями. Это можно сделать так:

```
int a = 1 << (c1 + 1);  // нули с 1 в позиции (c1 + 1)
int b = a - 1;          // нули, за которыми следуют c1 + 1 единиц
int c = b << (c0 - 1);  // c1+1 единиц, за которыми следуют c0 - 1 нулей.
n |= c;
```

Далее приведена реализация `getPrev`:

```
1  int getPrev(int n) {
2    int temp = n;
3    int c0 = 0;
4    int c1 = 0;
5    while (temp & 1 == 1) {
6      c1++;
7      temp >>= 1;
8    }
9
10   if (temp == 0)  return -1;
11
12   while (((temp & 1) == 0) && (temp != 0)) {
13     c0++;
14     temp >>= 1;
15   }
16
17   int p = c0 + c1;                // Позиция крайней правой незавершающей единицы
18   n &= ((~0) << (p + 1));         // Сброс битов, начиная с p
19
20   int mask = (1 << (c1 + 1)) - 1; // Последовательность из (c1+1) единиц
21   n |= mask << (c0 - 1);
22
23   return n;
24 }
```

Арифметическое решение для getNext

Если `c0` — количество завершающих нулей, `c1` — размер следующего блока единиц и `p = c0 + c1`, то наше решение можно сформулировать так:

1. Установить p-й бит в 1.
2. Сбросить все биты после p.
3. Установить биты с 0 до `c1 – 2` в 1 (всего `c1 – 1` битов).

Простой и быстрый способ выполнения шагов 1 и 2 — установить завершающие нули в 1 (получив p завершающих единиц), а затем прибавить 1. Сложение с единицей инвертирует завершающие единицы; таким образом, мы получим 1 в бите p, за которым идут p нулей. Можно решить эту задачу арифметически.

```
n += 2^c0 - 1;       // Установить завершающие 0 в 1, получив p завершающих 1
n += 1;              // Сбросить первые p единиц и поместить 1 в бит p
```

Теперь, чтобы выполнить шаг 3 арифметически, нужно:

```
n += 2^(c1 - 1) - 1;   // Установить завершающие c1 - 1 нулей в 1.
```

Все эти вычисления приводятся к следующему виду:

```
next = n + (2^c0 - 1) + 1 + (2^(c1 - 1) - 1) =
     = n + 2^c0 + 2^(c1 - 1) - 1
```

Лучше всего то, что при небольших манипуляциях с битами решение программируется очень просто:

```
1  int getNextArith(int n) {
2    /* ... c0 и c1 вычисляются так же, как и прежде ... */
3    return n + (1 << c0) + (1 << (c1 - 1)) - 1;
4  }
```

Арифметическое решение для getPrev

Если c_1 — количество конечных единиц, c_0 — размер блока соседних с ними нулей и $p = c_0 + c_1$, можно реализовать `getPrev` так:

1. Сбросить бит p.
2. Установить все биты после p в 1.
3. Сбросить биты с 0 по $c_0 - 1$.

Можно решить эту задачу арифметически. Предположим, что `n = 10000011`. Тогда $c_1 = 2$, $c_0 = 5$.

```
n -= 2^c1 - 1;       // Удалить завершающие единицы. n = 10000000.
n -= 1;              // Переключение завершающих нулей. n = 01111111.
n -= 2^(c0 - 1) - 1;   // Переключение последних (c0 - 1) нулей. n = 01110000.
```

В математической форме это сводится к следующему виду:

```
next = n - (2^c1 - 1) - 1 - (2^(c0 - 1) - 1)
     = n - 2^c1 - 2^(c0 - 1) + 1
```

И снова решение получается очень простым:

```
1  int getPrevArith(int n) {
2    /* ... c0 и c1 вычисляются так же, как и прежде ... */
3    return n - (1 << c1) - (1 << (c0 - 1)) + 1;
4  }
```

Уф! К счастью, никто не ждет, что вы сможете проделать все это на собеседовании, по крайней мере без значительной помощи со стороны интервьюера.

5.5. Объясните, что делает код: `((n & (n - 1)) == 0)`.

РЕШЕНИЕ

Будем действовать «от обратного».

Что означает `A & B == 0`?

Это означает, что `a` и `B` не содержат единичных битов на одних и тех же позициях. Если `n & (n-1) == 0`, то `n` и `n - 1` не имеют единиц в общих разрядах.

На что похоже `n - 1` (по сравнению с `n`)?

Попытайтесь проделать вычитание вручную (в двоичной или десятичной системе). Что произойдет?

```
  1101011000 [двоичн.]
-          1
= 1101010111 [двоичн.]
  593100 [десятичн.]
-      1
= 593099 [десятичн.]
```

При вычитании единицы проверяется младший бит; 1 заменяется на 0. Но, если младший бит содержит 0, придется «заимствовать» из старшего бита. Так каждый бит последовательно заменяется с 0 на 1, пока вы не дойдете до 1. Затем вы инвертируете единицу в ноль, — все готово.

Таким образом, можно сказать, что `n - 1` будет совпадать с `n`, за исключением того, что младшим нулям в `n` соответствуют единицы в `n - 1`, а последний единичный бит в `n` становится нулем в `n - 1`:

```
n     = abcde1000
n - 1 = abcde0111
```

Что значит `n & (n - 1) == 0`?

`n` и `n - 1` не содержат единиц в общих разрядах. Предположим, они имеют вид:

```
n     = abcde1000
n - 1 = abcde0111
```

`abcde` должны быть нулевыми битами, то есть `n` имеет вид: `00001000`. Таким образом, значение `n` — степень двойки.

Итак, мы пришли к ответу: логическое выражение `((n & (n - 1)) == 0)` истинно, если `n` является степенью двойки или равно нулю.

5.6. Напишите функцию, определяющую количество битов, которые необходимо изменить, чтобы из целого числа `A` получить целое число `B`.

Пример:

Ввод: `29 (или: 11101), 15 (или: 01111)`

Вывод: 2

РЕШЕНИЕ

Эта сложная на первый взгляд задача на самом деле очень проста. Чтобы решить ее, нужно задать себе вопрос: «Как узнать, какие биты в двух числах различаются?» Ответ прост: с помощью операции.

Каждая единица результата `XOR` соответствует биту, который не совпадает в числах `A` и `B`. Следовательно, для получения количества несовпадающих битов в числах `a` и `B` достаточно подсчитать число единиц в `A^B`:

```
1  int bitSwapRequired(int a, int b) {
2    int count = 0;
3    for (int c = a ^ b; c != 0; c = c >> 1) {
4      count += c & 1;
5    }
6    return count;
7  }
```

Этот код работает, но его можно сделать еще лучше. Вместо многократного сдвига c для проверки значащего бита достаточно будет многократно инвертировать младший разряд и подсчитывать, сколько раз понадобится проделать эту операцию, пока число не станет равным нулю. Операция `c = c & (c - 1)` сбрасывает младший ненулевой бит числа `c`.

Следующий код реализует данный метод:

```
1  public static int bitSwapRequired(int a, int b) {
2    int count = 0;
3    for (int c = a ^ b; c != 0; c = c & (c - 1)) {
4      count++;
5    }
6    return count;
7  }
```

Это одна из типичных задач на работу с битами, которые встречаются на собеседованиях. Если вы никогда с ними не сталкивались, вам будет сложно сразу решить задачу, поэтому запомните использованные приемы — они вам пригодятся.

5.7. Напишите программу, меняющую местами четные и нечетные биты числа с минимальным количеством инструкций (например, меняются местами биты 0 и 1, биты 2 и 3 и т. д.).

РЕШЕНИЕ

Как и в предыдущих случаях, полезно взглянуть на задачу с другой стороны. Работать с парами битов по отдельности будет слишком сложно, да и не очень эффективно. Что можно сделать?

Можно решить задачу, выполняя сначала операции с нечетными, а потом с четными битами? Можем мы взять `n` и сдвинуть нечетные биты на `1`? Да. Создадим маску для всех нечетных битов — `10101010` (или `0xAA`), затем сместим их вправо на `1`, расположив на месте четных. (Для четных битов выполним такую же операцию.) Остается столько соединить эти два значения.

Получается всего 5 операций. Реализация этого алгоритма выглядит так:

```
1  int swapOddEvenBits(int x) {
2    return ( ((x & 0xaaaaaaaa) >>> 1) | ((x & 0x55555555) << 1) );
3  }
```

Обратите внимание на применение логического сдвига вправо (вместо арифметического). Это связано с тем, что знаковый бит должен заполняться нулем.

Приведенный выше код реализован на Java для 32-разрядных целых чисел. Если вы работаете с 64-разрядными числами, нужно изменить маску, но логика останется той же.

5.8. Содержимое монохромного экрана хранится в одномерном массиве байтов, так что в одном байте содержится информация о восьми соседних пикселах.

Ширина изображения `w` кратна 8 (то есть байты не переходят между столбцами). Высоту экрана можно рассчитать, зная длину массива и ширину экрана. Реализуйте функцию, которая рисует горизонтальную линию из точки `(x1, y)` в точку `(x2, y)`.

Сигнатура метода должна выглядеть примерно так:

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

РЕШЕНИЕ

«Наивное» решение задачи тривиально — в цикле `for` перемещаться от `x1` до `x2`, устанавливая каждый пиксел на пути. Но оно не слишком интересно, верно? (Да и не слишком эффективно, кстати.)

При анализе ситуации можно прийти к выводу, что если `x1` и `x2` находятся далеко друг от друга, то между ними помещаются несколько полных байтов. Эти полные байты можно последовательно задать с помощью функции `screen[byte_pos] = 0xFF`. Начало и конец линии можно задать с помощью масок:

```
1  void drawLine(byte[] screen, int width, int x1, int x2, int y) {
2    int start_offset = x1 % 8;
3    int first_full_byte = x1 / 8;
4    if (start_offset != 0) {
5      first_full_byte++;
6    }
7
8    int end_offset = x2 % 8;
9    int last_full_byte = x2 / 8;
10    if (end_offset != 7) {
11      last_full_byte--;
12    }
13
14    // Запись полных байтов
15    for (int b = first_full_byte; b <= last_full_byte; b++) {
16      screen[(width / 8) * y + b] = (byte) 0xFF;
17    }
18
19    // Создание масок для начала и конца линии
20    byte start_mask = (byte) (0xFF >> start_offset);
21    byte end_mask = (byte) ~(0xFF >> (end_offset + 1));
22
23    // Назначение начала и конца линии
24    if ((x1 / 8) == (x2 / 8)) { // x1 и x2 находятся в одном байте
25      byte mask = (byte) (start_mask & end_mask);
26      screen[(width / 8) * y + (x1 / 8)] |= mask;
27    } else {
28      if (start_offset != 0) {
29        int byte_number = (width / 8) * y + first_full_byte - 1;
30        screen[byte_number] |= start_mask;
31      }
32      if (end_offset != 7) {
33        int byte_number = (width / 8) * y + last_full_byte + 1;
34        screen[byte_number] |= end_mask;
35      }
36    }
37  }
```

Будьте осторожны при решении подобной задачи: в ней кроется немало ловушек и особых случаев. Например, необходимо предусмотреть ситуацию, когда `x1` и `x2` находятся в одном байте. Только самые внимательные кандидаты могут реализовать этот код без ошибок.

6

Головоломки

6.1. Есть 20 баночек с таблетками. В 19 баночках лежат таблетки весом 1 г, а в одной — весом 1,1 г. Даны весы, показывающие точный вес. Как за одно взвешивание найти банку с тяжелыми таблетками?

РЕШЕНИЕ

Иногда хитроумные ограничения могут стать подсказкой. В нашем случае подсказка спрятана в ограничении, разрешающем использовать весы только один раз.

У нас только одно взвешивание, а это значит, что придется одновременно взвешивать много таблеток. Фактически, мы должны одновременно взвесить таблетки из всех 19 банок. Если пропустить две (или более) банки, как их различить? Не забывайте: взвешивание только *одно*!

Как же взвесить несколько банок и понять, в какой из них находятся «дефектные» таблетки? Давайте представим, что у нас есть только две банки, в одной из них лежат более тяжелые таблетки. Если взять по одной таблетке из каждой банки и взвесить их одновременно, то общий вес будет 2,1 г, но при этом мы не узнаем, какая из банок дала дополнительные 0,1 г. Значит, нужно взвешивать как-то иначе.

Если мы возьмем одну таблетку из банки № 1 и две из банки № 2, то что покажут весы? Результат зависит от веса таблеток. Если банка № 1 содержит более тяжелые таблетки, то вес будет 3,1 г. Если с тяжелыми таблетками банка № 2 — то 3,2 грамма. В этом и кроется ключ к задаче.

Нам известен ожидаемый вес таблеток. Разница между ожидаемым и фактическим весом покажет, какая банка содержит более тяжелые таблетки — при условии, что мы взяли разное количество таблеток из каждой банки.

Этот подход можно обобщить: возьмем одну таблетку из банки № 1, две таблетки из банки № 2, три таблетки из банки № 3 и т. д. Взвесьте этот набор таблеток. Если все таблетки весят 1 г, то результат составит 210 г. Весь «излишек» должен быть внесен за счет банки с тяжелыми таблетками.

Таким образом, номер банки можно узнать по простой формуле: `(вес - 210) / 0,1`. Если суммарный вес таблеток составляет 211,3 г, то тяжелые таблетки находились в банке № 13.

6.2. Имеется баскетбольное кольцо, и вам предлагается сыграть в одну из двух игр:

Игра 1: У вас есть один бросок в кольцо.

Игра 2: У вас есть три попытки, но для победы потребуется не менее двух попаданий.

Если p — вероятность попадания в кольцо при броске, при каких значениях p следует выбирать ту или иную игру?

РЕШЕНИЕ

Для решения этой задачи можно напрямую воспользоваться законами теории вероятностей и сравнить вероятность победы в каждом случае.

Вероятность победы в игре 1

Вероятность победы по определению равна p.

Вероятность победы в игре 2

Пусть `s(k, n)` — вероятность k точных попаданий из n попыток. Вероятность победы в игре 2 является вероятностью того, что из трех попыток успешными будут две ИЛИ три. Другими словами:

```
P(победа) = s(2,3) + s(3,3)
```

Вероятность трех попаданий:

$s(3,3) = p^3$

Вероятность двух удачных попаданий:

```
P(попадание 1 и 2, промах 3) +
    + P(попадание 1 и 3, промах 2) +
    + P(промах 1, попадание 2 и 3) =
= p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p
= 3 (1 - p) p²
```

Собрав все вместе, мы получим:

```
= p³ + 3 (1 - p) p²
= p³ + 3p² - 3p³
= 3p² - 2p³
```

Какой вариант игры выбрать?

Правила игры 1 выбираются в том случае, если `P(игра 1) > P(игра 2)`:

```
p > 3p² - 2p³.
1 > 3p - 2p²
2p² - 3p + 1 > 0
(2p - 1)(p - 1) > 0
```

Оба множителя могут быть либо положительными, либо отрицательными. Но мы знаем, что `p < 1`, а следовательно, `p - 1 < 0`. Это говорит о том, что оба значения должны быть отрицательными.

```
2p - 1 < 0
2p < 1
p < .5
```

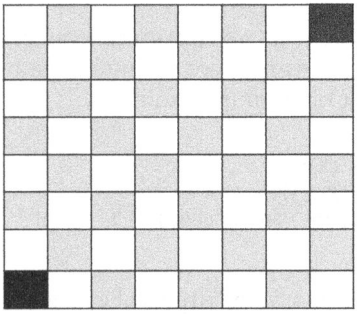
Итак, игра 1 выбирается в том случае, если `p < 0,5`. Если же `p = 0, 0.5` или `1`, значит, `P(игра 1) = P(игра 2)`, поэтому выбор не имеет значения.

6.3. Имеется шахматная доска размером 8×8, из которой были вырезаны два противоположных по диагонали угла, и 31 кость домино; каждая кость домино может закрыть два квадратика на поле. Можно ли вымостить костями всю доску? Обоснуйте ответ (приведите пример или докажите, что это невозможно).

РЕШЕНИЕ

На первый взгляд кажется, что это возможно. Доска имеет размеры 8×8; из 64 клеток исключаются 2, остается 62. Наверное, разложить 31 кость как-нибудь удастся?

Когда мы попытаемся разложить домино в первом ряду, то в нашем распоряжении будет только 7 квадратов, одна кость переходит на второй ряд. Затем мы размещаем домино во втором ряду, и опять одна кость переходит на третий ряд.



В каждом ряду всегда будет оставаться одна кость, которую нужно перенести на следующий ряд. Не имеет значения, сколько вариантов раскладки мы опробуем, разложить все кости все равно не получится.

Существует другое, более основательное доказательство. Шахматная доска делится на 32 черные и 32 белые клетки. Удаляя противоположные углы (обратите внимание, что эти клетки окрашены в один и тот же цвет), мы оставляем 30 клеток одного и 32 клетки другого цвета. Предположим, что теперь у нас есть 30 черных и 32 белых квадрата.

Каждая кость, которую мы будем класть на доску, будет занимать одну черную и одну белую клетку. Поэтому 31 кость домино займет 31 белую и 31 черную клетку. Но на нашей доске всего 30 черных и 32 белые клетки, поэтому разложить кости невозможно.

6.4. На каждой из трех вершин треугольника сидит муравей. Какова вероятность столкновения (на любой из сторон), если муравьи начнут ползти по сторонам треугольника? Предполагается, что каждый муравей выбирает направление случайным образом, вероятность выбора направлений одинакова и все муравьи ползут с одинаковой скоростью.

Также определите вероятность столкновения для n муравьев на многоугольнике с n вершинами.

РЕШЕНИЕ

Столкновения возможны только в том случае, если два муравья движутся навстречу друг другу. Следовательно, единственный способ избежать столкновений — двигаться в одном направлении (по часовой или против часовой стрелки). Мы вычислим эту вероятность и будем двигаться в обратном направлении.

Так как каждый муравей может двигаться в двух направлениях, а муравьев всего три, вероятность составит:

```
P(по часовой стрелке) = (½)³
P(против часовой стрелки) = (½)³
P(в одном направлении) = (½)³ + (½)³ = ¼
```

Вероятность столкновения муравьев теперь можно представить как вероятность того, что муравьи *не будут* двигаться в одном направлении:

```
P(столкновение) = 1 - P(в том же направлении) = 1 - ¼ = ¾
```

Обобщим это решение для случая n-угольника: существует только два маршрута, по которым муравьи могут двигаться без столкновений, но возможных маршрутов теперь 2^n. Общая вероятность столкновения:

```
P(по часовой стрелке) = (½)ⁿ
P(против часовой стрелки) = (½)ⁿ
P(в одном направлении) = 2(½)ⁿ (½)ⁿ = (½)ⁿ⁻¹
P(столкновение) = 1 - P(в одном направлении) = 1 - (½)ⁿ⁻¹
```

6.5. У вас есть пятилитровый и трехлитровый кувшины и неограниченное количество воды. Как отмерить ровно 4 литра воды? Кувшины имеют неправильную форму, поэтому точно отмерить «половину» кувшина невозможно.

РЕШЕНИЕ

Если мы поэкспериментируем с кувшинами, то обнаружим, что можно отмерить 4 л, если переливать воду из кувшина в кувшин следующим образом:

5 л	3 л	Примечание
5	0	Заполнили 5-литровый кувшин
2	3	Содержимое 5-литрового кувшина переливается в 3-литровый
0	2	Вылили 3 литра, перелили из кувшина в кувшин 2 литра
5	2	Заполнили 5-литровый кувшин
4	3	Заполнили 3-литровый кувшин, долив воду из 5-литрового
4		Готово! Мы получили 4 литра

Эта головоломка, как и многие остальные, имеет математическую основу. Если размеры двух кувшинов являются взаимно простыми числами, то с их помощью можно найти последовательность переливаний для любого значения объема от 1 до суммарного объема кувшинов.

6.6. На остров приезжает гонец со странным приказом: все голубоглазые люди должны как можно быстрее покинуть остров. Самолет улетает каждый вечер в 20:00. Каждый человек может видеть цвет глаз других, но не знает цвет собственных (и никто не имеет права сказать человеку, какой у него цвет глаз). Жители острова не знают, сколько на нем живет голубоглазых; известно лишь то, что есть хотя бы один. Сколько дней потребуется, чтобы все голубоглазые уехали?

РЕШЕНИЕ

Воспользуемся методом «базового случая с расширением». Предположим, что на острове находится `n` людей и `c` из них — голубоглазые. В условиях задачи явно указано, что `c > 0`.

c = 1: у одного человека голубые глаза

Предположим, что все люди на острове достаточно умны. Если известно, что на острове есть только одни голубоглазый человек, то, обнаружив, что у всех глаза не голубые, он придет к выводу, что он и является тем единственным голубоглазым человеком, которому следует улететь вечерним рейсом.

c = 2: у двух человек голубые глаза

Два человека с голубыми глазами видят друг друга, но не знают, чему равно `c`: `c = 1` или `c = 2`. Из предыдущего случая известно, что если `c = 1`, то голубоглазый человек может себя идентифицировать и покинуть остров в первый же вечер. Если голубоглазый человек находится на острове (`c = 2`), это означает, что человек, видящий только одного голубоглазого, сам голубоглазый. Оба человека должны будут вечером покинуть остров.

c > 2: общий случай

С увеличением `c` действует та же логика. Если `c = 3`, то эти три человека сразу увидят, что на острове есть еще два или три человека с голубыми глазами. Если бы таких людей было двое, они покинули бы остров накануне. Поскольку на острове все еще остаются голубоглазые люди, то любой человек может прийти к заключению, что `c = 3` и что у него голубые глаза. Все они уедут той же ночью.

Эта схема рассуждений обобщается для произвольного значения `c`. Поэтому если на острове находится `c` человек с голубыми глазами, понадобится `c` ночей, чтобы все они покинули остров. При этом все они покинут остров одновременно.

6.7. Королева нового постапокалиптического мира обеспокоена проблемой рождаемости. Она издает закон, по которому в каждой семье должна родиться хотя бы одна девочка. Если все семьи повинуются закону, то есть заводят детей,

пока не родится девочка, после чего немедленно прекращают, — каким будет соотношение полов в новом поколении? (Предполагается, что вероятности рождения мальчика или девочки равны.) Сначала решите задачу на логическом уровне, а затем напишите компьютерную модель.

РЕШЕНИЕ

Если каждая семья следует этой политике, то в ней рождается серия из 0 или более мальчиков, за которыми следует одна девочка. Иначе говоря, последовательность имеет следующий вид (*G* — девочка, *B* — мальчик): *G*; *BG*; *BBG*; *BBBG*; *BBBBG* и т. д.

У задачи есть несколько решений.

Математическое решение

Можно рассчитать вероятность каждой последовательности:

- `P(G)` = 1/2. Другими словами, у 50% семей сразу рождается девочка, остальные продолжают попытки.
- `P(BG)` = 1/4. У семей, заводящих второго ребенка (таких 50%), с вероятностью 50% рождается девочка.
- `P(BBG)` = 1/8. У семей, заводящих третьего ребенка (25%), с вероятностью 50% рождается девочка.

И так далее.

Мы знаем, что в каждой семье рождается ровно одна девочка. Сколько в среднем мальчиков будет в каждой семье? Чтобы ответить на этот вопрос, следует рассмотреть математическое ожидание количества мальчиков. Оно равно сумме вероятностей каждой последовательности, умноженных на количество мальчиков в этой последовательности.

Последовательность	Количество мальчиков	Вероятность	Количество мальчиков * * Вероятность
G	0	1/2	0
BG	1	1/4	1/4
BBG	2	1/8	2/8
BBBG	3	1/16	3/16
BBBBG	4	1/32	4/32
BBBBBG	5	1/64	5/64
BBBBBBG	6	1/128	6/128

Другими словами, вычисляемая величина равна

$$\sum_{i=0}^{\infty} \frac{i}{2^i}$$

Вряд ли вы с ходу выдадите ответ, но можно попытаться дать оценку. Приведем члены ряда к общему знаменателю 128 (2^6):

$$1/4 = 32/128$$
$$2/8 = 32/128$$

3/16 = 24/128

4/32 = 16/128

5/64 = 10/128

6/128 = 6/128

(32 + 32 + 24 + 16 + 10 + 6)/128 = 120/128

Похоже, сумма сходится к 128/128 (то есть к 1). Конечно, интуитивное «похоже» весьма ценно, но вряд ли оно сойдет за математическую концепцию. Тем не менее это подсказка, и мы можем применить логику. Итак, должен ли ряд сходиться к 1?

Логическое решение

Если приведенная сумма равна 1, это означает, что распределение полов оказывается практически равномерным. В каждой семье рождается ровно одна девочка и в среднем один мальчик. Таким образом, политика контроля рождаемости оказывается неэффективной.

На первый взгляд это кажется нелогичным. Политика должна способствовать рождению девочек, так как она гарантирует, что в каждой семье родится девочка.

С другой стороны, семьи, которые продолжают заводить детей, пополняют население мальчиками, а это может компенсировать влияние политики «одной девочки».

Представьте, что последовательности всех семей объединены в одну большую строку. Таким образом, если в семье 1 родились дети *BG*, в семье 2 — *BBG* и в семье 3 — *G*, строка имеет вид *BGBBGG*.

Собственно, группировка здесь несущественна, потому что нас интересует население в целом. С таким же успехом можно присоединять к строке пол каждого родившегося ребенка (*B* или *G*).

Какова вероятность того, что следующим символом в строке будет *G*? Вероятности рождения мальчиков и девочек одинаковы, поэтому вероятность того, что следующим символом будет *G*, равна 50%. Итак, примерно половина строки должна состоять из *G*, а другая половина из *B*, что дает равномерное распределение полов.

И этот вывод вполне логичен. Биология не изменилась; половину новорожденных составляют мальчики, а другую половину девочки. Правило, согласно которому семья в какой-то момент перестает заводить новых детей, этого факта не изменяет.

Итак, искомое соотношение полов — 50% девочек и 50% мальчиков.

Моделирование

Ниже приведена простая реализация, напрямую моделирующая условия задачи.

```
1  double runNFamilies(int n) {
2    int boys = 0;
3    int girls = 0;
4    for (int i = 0; i < n; i++) {
5      int[] genders = runOneFamily();
```

```
6       girls += genders[0];
7       boys += genders[1];
8     }
9     return girls / (double) (boys + girls);
10  }
11
12  int[] runOneFamily() {
13    Random random = new Random();
14    int boys = 0;
15    int girls = 0;
16    while (girls == 0) { // пока не родится девочка
17      if (random.nextBoolean()) { // девочка
18        girls += 1;
19      } else { // boy
20        boys += 1;
21      }
22    }
23    int[] genders = {girls, boys};
24    return genders;
25  }
```

И конечно, при достаточно больших значениях n вы получите результат, очень близкий к 0,5.

6.8. Имеется 100-этажное здание. Если яйцо сбросить с высоты *N*-го этажа (или с большей высоты), оно разобьется. Если его бросить с меньшего этажа, оно не разобьется. У вас есть два яйца; найдите *N* за минимальное количество бросков.

РЕШЕНИЕ

Независимо от того, с какого этажа мы бросаем яйцо № 1, при броске яйца № 2 придется использовать линейный поиск (от самого низкого до самого высокого этажа) между этажом, на котором яйцо разбилось, и следующим наивысшим этажом, при броске с которого яйцо останется целым. Например, если яйцо № 1 остается целым при падении с 5-го по 10-й этаж, но разбивается при броске с 15-го этажа, то яйцо № 2 придется (в худшем случае) сбрасывать с 11-го, 12-го, 13-го и 14-го этажей.

Метод

Для начала предположим, что яйцо бросается с 10-го этажа, потом с 20-го...

- Если яйцо № 1 разбилось на первом броске (этаж 10-й), то количество бросков заведомо не превысит 10.
- Если яйцо № 1 разбивается на последнем броске (этаж 100-й), тогда потребуется в худшем случае 19 бросков (этажи 10-й, 20-й, ..., 90-й, 100-й, затем с 91-го до 99-го).

Это хорошо, но мы рассмотрели безусловно худший случай. Чтобы добиться более эффективного результата, следует заняться «усреднением».

Наша цель — создать систему броска яйца №1, при которой количество бросков остается по возможности постоянным, а яйцо №1 разбивается при первом или при последнем броске.

1. Идеально сбалансированной будет та система, в которой значение `Drops(Egg1) + + Drops(Egg2)` остается постоянным независимо от того, на каком этаже разбилось яйцо № 1.
2. Если это условие выполняется, каждый «лишний» бросок яйца № 1 оставляет на один бросок меньше для яйца № 2.
3. Следовательно, каждый бросок сокращает на 1 количество бросков, потенциально необходимых яйцу № 2. Если яйцо № 1 бросается сначала с 20-го, а потом с 30-го этажа, то яйцу № 2 понадобится не более 9 бросков. Когда мы бросаем яйцо № 1 в очередной раз, то должны уменьшить количество бросков яйца № 2 до 8. Для этого достаточно бросить яйцо № 1 с 39-го этажа.
4. Следовательно, броски яйца № 1 должны начаться с этажа `X`, затем подняться на `X-1` этажей, затем на `X-2` этажей, пока не будет достигнут 100-й этаж.
5. Решение описывается следующей формулой:

$$X + (X-1) + (X-2) + ... + 1 = 100$$

$$X(X+1)/2 = 100$$

$$X \sim 13{,}65$$

Очевидно, значение `X` должно быть целым. В какую сторону округлять: вверх или вниз?

- Если округлить `X` вверх до 14, то мы сначала поднимаемся на 14-й этаж, потом на 13-й, 12-й и т. д. Последнее приращение будет равно 4 и произойдет на этаже 99. Если яйцо № 1 разбилось на одном из предыдущих этажей, мы знаем, что система сбалансирована, потому что сумма количества бросков яиц № 1 и № 2 всегда равна 14. Если яйцо № 1 не разбилось к 99-му этажу, тогда нам понадобится еще один бросок, чтобы узнать, разобьется ли оно на 100-м этаже. В любом случае количество бросков не превысит 14.
- Если округлить `X` вниз до 13, то броски будут выполняться на этажах 13, 12, 11 и т. д. Последнее приращение будет равно 1 и произойдет на этаже 91. Таким образом, после 13 бросков этажи с 92-го по 100-й еще не были исследованы. Проверить эти этажи за один бросок не удастся (а ведь это было бы необходимо для простой «ничьей»).

Итак, `X` следует округлять до 14. Мы сначала попадаем на 14-й этаж, затем на 27-й, затем на 39-й и т. д. Получается, что в худшем случае задача решается за 14 шагов.

Как и в других задачах максимизации/минимизации, ключом к решению является «балансировка худшего случая».

Описанный метод смоделирован в следующем коде.

```
1  int breakingPoint = ...;
2  int countDrops = 0;
3
4  boolean drop(int floor) {
5    countDrops++;
6    return floor >= breakingPoint;
7  }
8
9  int findBreakingPoint(int floors) {
```

```
10    int interval = 14;
11    int previousFloor = 0;
12    int egg1 = interval;
13
14    /* Бросать egg1 с уменьшающимися интервалами. */
15    while (!drop(egg1) && egg1 <= floors) {
16      interval -= 1;
17      previousFloor = egg1;
18      egg1 += interval;
19    }
20
21    /* Бросать egg2 с приращением в 1 этаж. */
22    int egg2 = previousFloor + 1;
23    while (egg2 < egg1 && egg2 <= floors && !drop(egg2)) {
24      egg2 += 1;
25    }
26
27    /* If it didn't break, return -1. */
28    return egg2 > floors ? -1 : egg2;
29  }
```

Если мы хотим обобщить этот код для другого количества этажей, решите для `x` следующее уравнение:

```
x(x+1)/2 = количество_этажей
```

6.9. В длинном коридоре расположены 100 закрытых замков. Человек сначала открывает все сто. Затем он закрывает каждый второй замок. Затем он делает еще один проход — «переключает» каждый третий замок (если замок был открыт, то он его закрывает, и наоборот). Процесс продолжается 100 раз, на *i*-м проходе изменяется состояние каждого *i*-го замка. Сколько замков останутся открытыми после 100-го прохода, когда «переключается» только замок № 100?

РЕШЕНИЕ

Сначала нужно определиться с тем, что такое «переключение». Это поможет нам разобраться, какие замки останутся открытыми в самом конце.

Вопрос: на каком проходе происходит переключение (открытие или закрытие)?

Замок `n` переключается один раз для каждого множителя `n` (включая `n` и 1). Таким образом, замок № 15 переключается на 1-м, 3-м, 5-м и 15-м проходах.

Вопрос: когда замок открыт?

Замок открыт, если значение счетчика (назовем его `x`) нечетное. Это можно использовать, разделив множители на «открытые» и «закрытые». Если остается непарный множитель, дверь останется открытой.

Вопрос: когда x будет нечетным?

Значение `x` будет нечетным, если `n` — квадрат числа. Сопоставим множители `n` с их дополнениями. Например, если `n = 36`, то мы получаем коэффициенты `(1, 36)`, `(2, 18)`, `(3, 12)`, `(4, 9)`, `(6, 6)`. Обратите внимание, что `(6, 6)` дает только один множитель, таким образом, мы получаем нечетное количество множителей.

Вопрос: сколько квадратов в нашей задаче?

В этой задаче 10 квадратов. Их можно явно пересчитать `(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)` или просто взять числа от 1 до 10 и возвести их в квадрат:

```
1*1, 2*2, 3*3, ..., 10*10
```

Таким образом, после всех обходов 10 замков останутся открытыми.

6.10. Имеется 1000 бутылок лимонада, ровно одна из которых отравлена. Также у вас есть 10 тестовых полосок для обнаружения яда. Даже одна капля яда окрашивает полоску и делает ее непригодной для дальнейшего использования. На тестовую полоску можно одновременно нанести любое количество капель, и одна полоска может использоваться сколько угодно раз (при условии, что все пробы были отрицательными). Однако вы можете проводить испытания не чаще одного раза в день, а для получения результата с момента проведения проходит семь дней. Как найти отравленную бутылку за минимальное количество дней?

Дополнительно

Напишите программную модель вашего решения.

РЕШЕНИЕ

Обратите внимание на формулировку задачи. Почему именно семь дней? Почему результаты тестирования не возвращаются немедленно?

Скорее всего, задержка между началом теста и чтением результатов означает, что в это время будет происходить что-то другое (проведение дополнительных тестов). Запомним это обстоятельство, но начнем с простого приема, который поможет нам лучше разобраться в сути задачи.

Наивное решение (28 дней)

Простое решение — распределить бутылки между 10 тестовыми полосками группами по 100, а затем подождать 7 дней. При получении результатов ищется полоска с положительным результатом, выбираются связанные с ней бутылки, и процесс повторяется. Операция выполняется до тех пор, пока в тестовом наборе не останется только одна бутылка.

1. Разделить бутылки между тестовыми полосками, нанести по одной капле на полоску.
2. Через 7 дней проверить результаты на полосках.

3. Для полоски с положительным результатом: выделить бутылки, связанные с ней, в новый набор. Если размер набора равен 1, отравленная бутылка найдена. Если размер больше 1, вернуться к шагу 1.

Чтобы смоделировать это решение, мы создадим классы `Bottle` и `TestStrip`, воспроизводящие функциональность задачи.

```
class Bottle {
  private boolean poisoned = false;
  private int id;

  public Bottle(int id) { this.id = id; }
  public int getId() { return id; }
  public void setAsPoisoned() { poisoned = true; }
  public boolean isPoisoned() { return poisoned; }
}

class TestStrip {
  public static int DAYS_FOR_RESULT = 7;
  private ArrayList<ArrayList<Bottle>> dropsByDay =
    new ArrayList<ArrayList<Bottle>>();
  private int id;

  public TestStrip(int id) { this.id = id; }
  public int getId() { return id; }

  /* Увеличить список дней/капель до нужного размера. */
  private void sizeDropsForDay(int day) {
    while (dropsByDay.size() <= day) {
      dropsByDay.add(new ArrayList<Bottle>());
    }
  }

  /* Добавить каплю из бутылки в определенный день. */
  public void addDropOnDay(int day, Bottle bottle) {
    sizeDropsForDay(day);
    ArrayList<Bottle> drops = dropsByDay.get(day);
    drops.add(bottle);
  }

  /* Проверить, есть ли в наборе отравленная бутылка. */
  private boolean hasPoison(ArrayList<Bottle> bottles) {
    for (Bottle b : bottles) {
      if (b.isPoisoned()) {
        return true;
      }
    }
    return false;
  }

  /* Получить бутылки, протестированные DAYS_FOR_RESULT дней назад. */
  public ArrayList<Bottle> getLastWeeksBottles(int day) {
    if (day < DAYS_FOR_RESULT) {
      return null;
    }
    return dropsByDay.get(day - DAYS_FOR_RESULT);
  }
```

```
51
52    /* Проверить отравленные бутылки, начиная с DAYS_FOR_RESULT */
53    public boolean isPositiveOnDay(int day) {
54      int testDay = day - DAYS_FOR_RESULT;
55      if (testDay < 0 || testDay >= dropsByDay.size()) {
56        return false;
57     }
58      for (int d = 0; d <= testDay; d++) {
59        ArrayList<Bottle> bottles = dropsByDay.get(d);
60        if (hasPoison(bottles)) {
61          return true;
62      }
63     }
64      return false;
65    }
66  }
```

Это всего лишь один из многих возможных способов моделирования поведения бутылок и тестовых полосок. У каждого способа есть свои достоинства и недостатки.

При наличии готовой инфраструктуры код тестирования нашего метода может быть реализован следующим образом:

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    int today = 0;
3
4    while (bottles.size() > 1 && strips.size() > 0) {
5      /* Выполнение тестов. */
6      runTestSet(bottles, strips, today);
7
8      /* Ожидание результатов. */
9      today += TestStrip.DAYS_FOR_RESULT;
10
11      /* Проверка результатов. */
12      for (TestStrip strip : strips) {
13        if (strip.isPositiveOnDay(today)) {
14          bottles = strip.getLastWeeksBottles(today);
15          strips.remove(strip);
16          break;
17      }
18     }
19    }
20
21    if (bottles.size() == 1) {
22      return bottles.get(0).getId();
23    }
24    return -1;
25  }
26
27  /* Равномерное распределение бутылок между тестовыми полосками. */
28  void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int
day) {
29    int index = 0;
30    for (Bottle bottle : bottles) {
31      TestStrip strip = strips.get(index);
32      strip.addDropOnDay(day, bottle);
33      index = (index + 1) % strips.size();
```

```
34    }
35  }
36
37  /* Полный код содержится в архиве примеров*/
```

Эта реализация предполагает, что в каждом круге тестирования доступно более одной тестовой полоски (данное предположение действительно для 1000 бутылок и 10 тестовых полосок).

Если сделать такое предположение нельзя, можно применить гарантированно надежный способ: если остается всего одна тестовая полоска, мы начинаем тестировать по одной бутылке: тестируем, ждем неделю, тестируем другую бутылку. Этот метод займет не более 28 дней.

Оптимизированное решение (10 дней)

Как упоминалось в начале решения, будет эффективнее проводить несколько тестов одновременно. Если разделить бутылки на 10 групп (бутылки 0–99 для полоски 0, бутылки 100–199 для полоски 1, бутылки 200–299 для полоски 2 и т. д.), то день 7 откроет первую цифру номера бутылки. Положительный результат полоски `i` в день 7 означает, что первая цифра (сотни) номера искомой бутылки равна `i`.

Другой способ деления бутылок может открыть вторую или третью цифру. Нам лишь нужно проводить тесты в разные дни, чтобы не путаться с результатами.

	Дни 0->7	Дни 1->8	Дни 2->9
Полоска 0	0xx	x0x	xx0
Полоска 1	1xx	x1x	xx1
Полоска 2	2xx	x2x	xx2
Полоска 3	3xx	x3x	xx3
Полоска 4	4xx	x4x	xx4
Полоска 5	5xx	x5x	xx5
Полоска 6	6xx	x6x	xx6
Полоска 7	7xx	x7x	xx7
Полоска 8	8xx	x8x	xx8
Полоска 9	9xx	x9x	xx9

Например, если в день 7 был получен положительный результат на полоске 4, в день 8 — положительный результат на полоске 3, а в день 9 — положительный результат на полоске 8, то это будет соответствовать бутылке 438.

Это решение в основном работает, если не считать одного граничного случая: что произойдет, если в номере отравленной бутылки повторяются цифры (например, 882 или 383)?

На самом деле эти случаи сильно различаются. Если в день 8 нет «новых» положительных результатов, можно заключить, что цифра 2 равна цифре 1.

Более серьезная проблема связана с тем, что произойдет, если в день 9 не будет новых положительных результатов. В этом случае мы знаем только то, что цифра 3 равна цифре 1 или цифре 2. Различить бутылки 383 и 388 не удастся, они дают одинаковый набор результатов тестирования.

Придется провести один дополнительный тест. Можно сделать это в конце, чтобы устранить неоднозначность, но также можно провести тест в день 3 просто на случай появления неоднозначности. Нужно лишь сдвинуть последнюю цифру, чтобы она оказалась в другом месте, чем результат дня 2.

	Дни 0->7	Дни 1->8	Дни 2->9	Дни 3->10
Полоска 0	0xx	x0x	xx0	xx9
Полоска 1	1xx	x1x	xx1	xx0
Полоска 2	2xx	x2x	xx2	xx1
Полоска 3	3xx	x3x	xx3	xx2
Полоска 4	4xx	x4x	xx4	xx3
Полоска 5	5xx	x5x	xx5	xx4
Полоска 6	6xx	x6x	xx6	xx5
Полоска 7	7xx	x7x	xx7	xx6
Полоска 8	8xx	x8x	xx8	xx7
Полоска 9	9xx	x9x	xx9	xx8

Теперь для бутылки 383 будут получены результаты (день 7 = 3, день 8 -> 8, день 9 -> [нет], день 10 -> 4), а для бутылки 388 — (день 7 = 3, день 8 -> 8, день 9 -> [нет], день 10 -> 9). Чтобы различить их, достаточно «инвертировать» сдвиги в результатах дня 10.

Что произойдет, если в день 10 новые результаты так и не появятся? Такое возможно?

Вообще-то возможно. Для бутылки 898 будут получены результаты (день 7 = 8, день 8 -> 9, день 9 -> [нет], день 10 -> [нет]). Однако в этом нет ничего ужасного, ведь нам нужно отличить бутылку 898 от 899. Для бутылки 899 будет получен результат (день 7 = 8, день 8 -> 9, день 9 -> [нет], день 10 -> 0). «Неоднозначным» бутылкам со дня 9 всегда будут соответствовать разные значения в день 10. Логика выглядит так:

- Если тест день 3->10 открывает новый результат, «отменить сдвиг» этого значения для получения третьей цифры.

В противном случае мы знаем, что третья цифра равна либо первой, либо второй, а третья при сдвиге также равна первой или второй. Следовательно, необходимо только понять, «сдвигается» первая цифра на вторую или наоборот. В первом случае третья цифра равна первой, а во втором — третья цифра равна второй.

Для правильной реализации этой логики придется действовать очень внимательно, чтобы избежать ошибок.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    if (bottles.size() > 1000 || strips.size() < 10) return -1;
3
4    int tests = 4; // Три цифры плюс одна дополнительная
5    int nTestStrips = strips.size();
6
7    /* Выполнение тестов. */
8    for (int day = 0; day < tests; day++) {
9      runTestSet(bottles, strips, day);
10   }
```

```
11
12    /* Получение результатов. */
13    HashSet<Integer> previousResults = new HashSet<Integer>();
14    int[] digits = new int[tests];
15    for (int day = 0; day < tests; day++) {
16      int resultDay = day + TestStrip.DAYS_FOR_RESULT;
17      digits[day] = getPositiveOnDay(strips, resultDay, previousResults);
18      previousResults.add(digits[day]);
19    }
20
21    /* Если результаты дня 1 совпадают с днем 0, обновить цифру. */
22    if (digits[1] == -1) {
23      digits[1] = digits[0];
24    }
25
26    /* Если результаты дня 2 совпадают с днем 0 или 1, проверить день 3.
27     * День 3 совпадает с днем 2, но с увеличением на 1. */
28    if (digits[2] == -1) {
29      if (digits[3] == -1) { /* День 3 не дал нового результата */
30        /* Цифра 2 равна цифре 0 или цифре 1. С другой стороны, цифра 2
31         * при увеличении также равна цифре 0 или цифре 1. Это означает,
32         * что увеличенная цифра 0 совпадает с цифрой 1 или наоборот. */
33        digits[2] = ((digits[0] + 1) % nTestStrips) == digits[1] ?
34                    digits[0] : digits[1];
35      } else {
36          digits[2] = (digits[3] - 1 + nTestStrips) % nTestStrips;
37     }
38    }
39
40    return digits[0] * 100 + digits[1] * 10 + digits[2];
41  }
42
43  /* Выполнение набора тестов для заданного дня. */
44  void runTestSet(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips, int day) {
45    if (day > 3) return; // Работает только для 3 дней (цифр) + еще один.
46
47    for (Bottle bottle : bottles) {
48      int index = getTestStripIndexForDay(bottle, day, strips.size());
49      TestStrip testStrip = strips.get(index);
50      testStrip.addDropOnDay(day, bottle);
51    }
52  }
53
54  /* Получение полоски для заданной бутылки в заданный день. */
55  int getTestStripIndexForDay(Bottle bottle, int day, int nTestStrips) {
56    int id = bottle.getId();
57    switch (day) {
58      case 0: return id /100;
59      case 1: return (id % 100) / 10;
60      case 2: return id % 10;
61      case 3: return (id % 10 + 1) % nTestStrips;
62      default: return -1;
63    }
64  }
65
66  /* Получение результатов, положительных для заданного дня. */
67  int getPositiveOnDay(ArrayList<TestStrip> testStrips, int day,
```

```
68                      HashSet<Integer> previousResults) {
69    for (TestStrip testStrip : testStrips) {
70      int id = testStrip.getId();
71      if (testStrip.isPositiveOnDay(day) && !previousResults.contains(id)) {
72        return testStrip.getId();
73      }
74    }
75    return -1;
76  }
```

При таком подходе для получения результата в худшем случае потребуется 10 дней.

Оптимальный подход (7 дней)

Даже это решение можно дополнительно оптимизировать, чтобы получить результат всего за 7 дней. Конечно, дальнейшее улучшение уже невозможно.

Что собой представляет тестовая полоска? Двоичный индикатор «отравлено/не отравлено». Можно ли связать 1000 ключей с 10 двоичными значениями, чтобы каждому ключу соответствовала уникальная конфигурация значений? Да, конечно.

Возьмем каждый номер бутылки и рассмотрим его двоичное представление. Если в `i`-м разряде находится 1, капля содержимого этой бутылки наносится на тестовую полоску `i`. Заметим, что $2^{10} = 1024$, поэтому 10 полосок будет достаточно для 1024 бутылок.

После 7-дневного ожидания остается обработать результаты. Если результат тестовой полоски `i` положителен, в итоговом значении устанавливается `i`-й бит.

Обработка всех тестовых полосок дает идентификатор отравленной бутылки.

```
1  int findPoisonedBottle(ArrayList<Bottle> bottles, ArrayList<TestStrip> strips) {
2    runTests(bottles, strips);
3    ArrayList<Integer> positive = getPositiveOnDay(strips, 7);
4    return setBits(positive);
5  }
6
7  /* Нанесение содержимого бутылок на тестовые полоски */
8  void runTests(ArrayList<Bottle> bottles, ArrayList<TestStrip> testStrips) {
9    for (Bottle bottle : bottles) {
10     int id = bottle.getId();
11     int bitIndex = 0;
12     while (id > 0) {
13       if ((id & 1) == 1) {
14         testStrips.get(bitIndex).addDropOnDay(0, bottle);
15       }
16       bitIndex++;
17       id >>= 1;
18     }
19   }
20 }
21
22 /* Получение результатов, положительных для заданного дня. */
23 ArrayList<Integer> getPositiveOnDay(ArrayList<TestStrip> testStrips, int day) {
24   ArrayList<Integer> positive = new ArrayList<Integer>();
25   for (TestStrip testStrip : testStrips) {
26     int id = testStrip.getId();
```

```
27      if (testStrip.isPositiveOnDay(day)) {
28        positive.add(id);
29      }
30    }
31    return positive;
32  }
33
34  /* Построение числа посредством установки битов по положительным результатам.
*/
35  int setBits(ArrayList<Integer> positive) {
36    int id = 0;
37    for (Integer bitIndex : positive) {
38      id |= 1 << bitIndex;
39    }
40    return id;
41  }
```

Такое решение работает при условии $2^T >= B$, где T — количество тестовых полосок, а B — количество бутылок.

7

Объектно-ориентированное проектирование

7.1. Разработайте структуры данных для универсальной колоды карт. Объясните, как разделить структуры данных на субклассы, чтобы реализовать игру в блэкджек.

РЕШЕНИЕ

Прежде всего необходимо определиться, что такое «универсальная колода карт». «Универсальной колодой» могут назвать как стандартную колоду карт для игры в покер, так и набор бейсбольных карточек или карт для игры «Уно». Очень важно уточнить у интервьюера, что значит «универсальная».

Предположим, интервьюер прояснил, что мы имеем дело со стандартной колодой из 52 карт, которая используется при игре в блэкджек или покер. Если это так, то решение будет выглядеть примерно так:

```
1  public enum Suit {
2    Club (0),  Diamond (1), Heart (2),  Spade (3);
3    private int value;
4    private Suit(int v) { value = v; }
5    public int getValue() { return value; }
6    public static Suit getSuitFromValue(int value) { ... }
7  }
8
9  public class Deck <T extends Card> {
10   private ArrayList<T> cards; // Все карты (сданные и нет)
11   private int dealtIndex = 0; // Маркер первой сданной карты
12
13   public void setDeckOfCards(ArrayList<T> deckOfCards) { ... }
14
15   public void shuffle() { ... }
16   public int remainingCards() {
17     return cards.size() - dealtIndex;
18   }
19   public T[] dealHand(int number) { ... }
20   public T dealCard() { ... }
21 }
22
23 public abstract class Card {
24   private boolean available = true;
25
26   /* Номинал карты - 2-10, 11 - валет, 12 - дама,
```

```
27     * 13 - король, 1 - туз */
28    protected int faceValue;
29    protected Suit suit;
30
31    public Card(int c, Suit s) {
32      faceValue = c;
33      suit = s;
34    }
35
36    public abstract int value();
37    public Suit suit() { return suit; }
38
39    /* Проверка доступности карты для сдачи */
40    public boolean isAvailable() { return available; }
41    public void markUnavailable() { available = false; }
42    public void markAvailable() { available = true; }
43  }
44
45  public class Hand <T extends Card> {
46    protected ArrayList<T> cards = new ArrayList<T>();
47
48    public int score() {
49      int score = 0;
50      for (T card : cards) {
51        score += card.value();
52     }
53      return score;
54    }
55
56    public void addCard(T card) {
57      cards.add(card);
58    }
59  }
```

В этом коде класс `Deck` реализован с применением обобщенных типов (generics), но тип `T` ограничивается `Card`. Также можно было реализовать `Card` как абстрактный класс, потому что методы вида `value()` не имеют особого смысла без связи с правилами конкретной игры. (Впрочем, на это можно возразить, что их все равно следует реализовать с использованием по умолчанию стандартных правил покера.)

Допустим, вы моделируете блэкджек, поэтому нам нужно знать номиналы карт. Фигурные карты имеют номинал 10, а туз — 11 (в большинстве случаев, впрочем, за этот нюанс отвечает класс `Hand`, а не следующий класс).

```
1  public class BlackJackHand extends Hand<BlackJackCard> {
2    /* У руки в блэкджеке может быть несколько разных значений,
3     * потому что тузы могут иметь разные номиналы. Вернуть наибольшее
4     * значение, меньшее 21, или наименьшее значение при переборе */
5    public int score() {
6      ArrayList<Integer> scores = possibleScores();
7      int maxUnder = Integer.MIN_VALUE;
8      int minOver = Integer.MAX_VALUE;
9      for (int score : scores) {
10       if (score > 21 && score < minOver) {
11         minOver = score;
12       } else if (score <= 21 && score > maxUnder) {
```

```
13          maxUnder = score;
14        }
15      }
16        return maxUnder == Integer.MIN_VALUE ? minOver : maxUnder;
17    }
18
19    /* Вернуть список всех возможных значений руки (с вариантами
20     * зачета каждого туза с 1 и 11 очками) */
21    private ArrayList<Integer> possibleScores() { ... }
22
23    public boolean busted() { return score() > 21; }
24    public boolean is21() { return score() == 21; }
25    public boolean isBlackJack() { ... }
26  }
27
28  public class BlackJackCard extends Card {
29    public BlackJackCard(int c, Suit s) { super(c, s); }
30    public int value() {
31      if (isAce()) return 1;
32      else if (faceValue >= 11 && faceValue <= 13) return 10;
33      else return faceValue;
34    }
35
36    public int minValue() {
37      if (isAce()) return 1;
38      else return value();
39    }
40
41    public int maxValue() {
42      if (isAce()) return 11;
43      else return value();
44    }
45
46    public boolean isAce() {
47      return faceValue == 1;
48    }
49
50    public boolean isFaceCard() {
51      return faceValue >= 11 && faceValue <= 13;
52    }
53  }
```

Это всего лишь один из возможных способов обработки тузов. Также можно было создать класс типа `Ace`, расширяющий `BlackJackCards`.

Полностью автоматизированная версия блэкджека содержится в архиве примеров.

7.2. Имеется центр обработки звонков с тремя уровнями сотрудников: оператор, менеджер и директор. Входящий телефонный звонок адресуется свободному оператору. Если оператор не может обработать звонок, он автоматически перенаправляется менеджеру. Если менеджер занят, то звонок перенаправляется директору. Разработайте классы и структуры данных для этой задачи. Реализуйте метод `dispatchCall()`, который перенаправляет звонок первому свободному сотруднику.

РЕШЕНИЕ

Каждая категория служащих выполняет свои обязанности, поэтому их специфические функции должны быть реализованы в их соответствующих классах.

Существует несколько общих полей, например адрес, имя, должность и возраст. Всю эту информацию имеет смысл хранить в одном классе, который может расширяться другими классами.

Наконец, нам понадобится класс `CallHandler`, который перенаправляет вызов конкретному человеку.

Учтите, что в объектно-ориентированном проектировании существует много вариантов архитектур объектов. Обсудите достоинства и недостатки разных вариантов с интервьюером. Как правило, проектировать следует с расчетом на долгосрочную гибкость и простоту сопровождения кода.

Давайте остановимся на каждом из классов поподробнее.

Класс `CallHanlder` представляет тело программы; через него проходят все поступающие звонки.

```
1  public class CallHandler {
2    /* 3 уровня работников: оператор, менеджер, директор. */
3    private final int LEVELS = 3;
4
5    /* Инициализация: 10 операторов, 4 менеджера, 2 директора. */
6    private final int NUM_RESPONDENTS = 10;
7    private final int NUM_MANAGERS = 4;
8    private final int NUM_DIRECTORS = 2;
9
10    /* Список работников по уровням.
11     * employeeLevels[0] = операторы
12     * employeeLevels[1] = менеджеры
13     * employeeLevels[2] = директора
14     */
15    List<List<Employee>> employeeLevels;
16
17    /* Очереди для каждого вызова */
18    List<List<Call>> callQueues;
19
20    public CallHandler() { ... }
21
22    /* Получение первого доступного работника, способного обработать звонок.*/
23    public Employee getHandlerForCall(Call call) { ... }
24
25    /* Звонок перенаправляется доступному работнику или сохраняется
26     * в очереди при отсутствии доступных работников. */
27    public void dispatchCall(Caller caller) {
28      Call call = new Call(caller);
29      dispatchCall(call);
30    }
31
32    /* Звонок перенаправляется доступному работнику или сохраняется
33     * в очереди при отсутствии доступных работников. */
34    public void dispatchCall(Call call) {
35      /* Попытка перенаправить звонок работнику с минимальным рангом. */
36      Employee emp = getHandlerForCall(call);
```

```
37      if (emp != null) {
38        emp.receiveCall(call);
39        call.setHandler(emp);
40      } else {
41        /* Звонок ставится в очередь в соответствии с его рангом. */
42        call.reply("Please wait for free employee to reply");
43        callQueues.get(call.getRank().getValue()).add(call);
44     }
45    }
46
47    /* Работник освободился. Если ему был назначен входящий звонок,
48     * вернуть true, если нет - вернуть false. */
49    public boolean assignCall(Employee emp) { ... }
50  }
```

Класс `Call` представляет звонок от абонента. Каждому звонку назначается минимальный ранг; этот звонок будет перенаправлен первому работнику, который сможет его обработать:

```
1  public class Call {
2    /* Минимальный ранг работника, который может обработать звонок. */
3    private Rank rank;
4
5    /* Абонент. */
6    private Caller caller;
7
8    /* Работник, обрабатывающий звонок. */
9    private Employee handler;
10
11    public Call(Caller c) {
12      rank = Rank.Responder;
13      caller = c;
14    }
15
16    /* Назначение работника для обработки звонка. */
17    public void setHandler(Employee e) { handler = e; }
18
19    public void reply(String message) { ... }
20    public Rank getRank() { return rank;  }
21    public void setRank(Rank r) { rank = r; }
22    public Rank incrementRank() { ... }
23    public void disconnect() { ...  }
24  }
```

`Employee` — суперкласс для классов `Director`, `Manager` и `Respondent`. Он реализуется в виде абстрактного класса, потому что создание экземпляров типа `Employee` не имеет смысла.

```
1  abstract class Employee {
2    private Call currentCall = null;
3    protected Rank rank;
4
5    public Employee(CallHandler handler) { ... }
6
7    /* Начало разговора. */
8    public void receiveCall(Call call) { ... }
9
```

```
    /* Проблема решена, разговор завершается */
    public void callCompleted() { ... }

    /* Проблема не решена. Звонок передается дальше
     * и назначается новому работнику. */
    public void escalateAndReassign() { ... }

    /* Новый вызов назначается работнику, если он свободен. */
    public boolean assignNewCall() { ... }

    /* Проверяет, свободен ли работник. */
    public boolean isFree() { return currentCall == null; }

    public Rank getRank() { return rank;  }
  }

```

Классы `Respondent`, `Director` и `Manager` представляют собой обычные расширения класса `Employee`.

```
class Director extends Employee {
  public Director() {
    rank = Rank.Director;
  }
}

class Manager extends Employee {
  public Manager() {
    rank = Rank.Manager;
  }
}

class Respondent extends Employee {
  public Respondent() {
    rank = Rank.Responder;
  }
}
```

Конечно, это только один из вариантов решения задачи. Есть и другие, ничуть не худшие.

Может показаться, что объем кода слишком велик для собеседования. Мы привели более развернутый код, чем реально необходимо. На реальном собеседовании некоторые детали стоит опустить — ими можно будет заняться при наличии свободного времени.

7.3. Разработайте модель музыкального автомата, используя принципы ООП.

РЕШЕНИЕ

Приступая к решению любых задач объектно-ориентированного проектирования, прежде всего задайте интервьюеру вопросы для прояснения ограничений. Будет этот музыкальный автомат работать с CD? Кассетами? MP3? Это компьютерная модель или речь идет о физическом устройстве? Должен ли автомат взимать плату? Если он принимает деньги, то в какой валюте? И должен ли выдавать сдачу?

К сожалению, в нашем распоряжении нет интервьюера, с которым можно поговорить. Вместо этого мы сделаем некоторые предположения. Допустим, что музыкальный автомат — это компьютерная модель, которая отражает работу физического устройства и работает бесплатно.

Давайте выделим базовые компоненты системы:

- Музыкальный автомат (`Jukebox`).
- Привод для проигрывания дисков (`CD`).
- Композиция (`Song`).
- Исполнитель (`Artist`).
- Список воспроизведения (`Playlist`).
- *Вывод* информации на экран (`Display`).

Возможные функции автомата:

- Создание списка воспроизведения.
- Выбор диска.
- Выбор композиции.
- Постановка композиции в очередь.
- Выбор следующей композиции из плей листа.

Пользователю также можно предоставить следующие дополнительные функции:

- Добавление композиций в очередь.
- Удаление их из очереди.
- Просмотр информации на экране.

Каждый из компонентов системы транслируется в объект, а каждое действие преобразуется в метод. Рассмотрим один из возможных вариантов реализации этого проекта.

Класс `Jukebox` представляет суть задачи. Большинство взаимодействий между компонентами системы или между системой и пользователем происходит в этом классе.

```
1  public class Jukebox {
2    private CDPlayer cdPlayer;
3    private User user;
4    private Set<CD> cdCollection;
5    private SongSelector ts;
6
7    public Jukebox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
8                   SongSelector ts) { ... }
9
10    public Song getCurrentSong() { return ts.getCurrentSong(); }
11    public void setUser(User u) { this.user = u;  }
12 }
```

Класс `CDPlayer`, как и настоящий проигрыватель компакт-дисков, может воспроизводить только один диск в определенный момент времени. Остальные диски хранятся в музыкальном автомате.

```
1  public class CDPlayer {
2    private Playlist p;
```

```
3    private CD c;
4
5    /* Конструкторы. */
6    public CDPlayer(CD c, Playlist p) { ... }
7    public CDPlayer(Playlist p) { this.p = p; }
8    public CDPlayer(CD c) { this.c = c; }
9
10    /* Воспроизведение песни */
11    public void playSong(Song s) { ... }
12
13    /* Get- и set-методы */
14    public Playlist getPlaylist() { return p; }
15    public void setPlaylist(Playlist p) { this.p = p; }
16
17    public CD getCD() { return c; }
18    public void setCD(CD c) { this.c = c; }
19  }
```

Класс `Playlist` управляет текущей и следующей композициями. Фактически это класс-обертка для очереди, содержащий ряд вспомогательных методов.

```
1  public class Playlist {
2    private Song song;
3    private Queue<Song> queue;
4    public Playlist(Song song, Queue<Song> queue) {
5     ...
6    }
7    public Song getNextSToPlay() {
8      return queue.peek();
9    }
10    public void queueUpSong(Song s) {
11      queue.add(s);
12    }
13  }
```

Классы `CD`, `Song` и `User` весьма просты. Они состоят из переменных экземпляра и `get`/`set`-методов.

```
1  public class CD { /* Идентификатор, исполнитель, композиции и т. д. */ }
2
3  public class Song { /* Идентификатор, CD, название, длина и т. д. */ }
4
5  public class User {
6    private String name;
7    public String getName() { return name; }
8    public void setName(String name) {  this.name = name; }
9    public long getID() { return ID; }
10    public void setID(long iD) { ID = iD; }
11    private long ID;
12    public User(String name, long iD) { ... }
13    public User getUser() { return this; }
14    public static User addUser(String name, long iD) { ... }
15  }
```

Данная реализация не является единственно правильной. Архитектура решения будет зависеть как от ответов интервьюера на ваши вопросы, так и от других ограничений.

7.4. Разработайте модель автостоянки, используя принципы ООП.

РЕШЕНИЕ

Формулировка этого задания выглядит весьма расплывчато, как это обычно бывает в реальном собеседовании. Прежде чем приступать к решению, нужно расспросить интервьюера о том, для какого транспорта предназначена стоянка, состоит ли она из нескольких уровней и т. д.

Сейчас мы ограничимся несколькими предположениями. Давайте слегка усложним задачу. (Если вы будете использовать собственные предположения, это будет замечательно.)

- Существует несколько уровней. На каждом уровне находится много парковочных мест.
- На парковке могут парковаться мотоциклы, автомобили и автобусы.
- На парковке есть мотоциклетные, компактные и большие парковочные места.
- Мотоцикл можно припарковать на любом месте.
- Автомобиль поместится на одно компактное или одно большое место.
- Автобусу потребуется пять больших мест, которые должны быть расположены последовательно в одном ряду. Автобус нельзя парковать на мотоциклетных и компактных парковочных местах.

В следующей реализации создается абстрактный класс `Vehicle`, от которого наследуют классы `Car`, `Bus` и `Motorcycle`. Для представления парковочных мест разного размера используется класс `ParkingSpot`, в котором есть переменная, указывающая размер места.

```
1  public enum VehicleSize { Motorcycle, Compact,   Large }
2
3  public abstract class Vehicle {
4    protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
5    protected String licensePlate;
6    protected int spotsNeeded;
7    protected VehicleSize size;
8
9    public int getSpotsNeeded() { return spotsNeeded; }
10    public VehicleSize getSize() { return size; }
11
12    /* Парковка на заданном месте  */
13    public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
14
15    /* Освобождение парковочных мест */
16    public void clearSpots() { ... }
17
18    /* Проверяет, хватит ли места для парковки (и свободно ли место).
19     * Проверяется только РАЗМЕР, но не количество мест. */
20    public abstract boolean canFitInSpot(ParkingSpot spot);
21 }
22
23 public class Bus extends Vehicle {
24    public Bus() {
25      spotsNeeded = 5;
26      size = VehicleSize.Large;
```

```
27    }
28
29    /* Проверяет, что место является большим (не количество мест.) */
30    public boolean canFitInSpot(ParkingSpot spot) { ... }
31  }
32
33  public class Car extends Vehicle {
34    public Car() {
35      spotsNeeded = 1;
36      size = VehicleSize.Compact;
37    }
38
39    /* Проверяет, что место является компактным или большим. */
40    public boolean canFitInSpot(ParkingSpot spot) { ... }
41  }
42
43  public class Motorcycle extends Vehicle {
44    public Motorcycle() {
45      spotsNeeded = 1;
46      size = VehicleSize.Motorcycle;
47    }
48
49    public boolean canFitInSpot(ParkingSpot spot) { ... }
50  }
```

Класс `ParkingLot` по сути является оберткой для массива `Levels`. Такая реализация позволяет отделить логику, связанную с фактическим нахождением свободных мест и парковкой автомобилей, от более общих действий `ParkingLot`. Если бы вы захотели использовать другую реализацию, информацию о парковочных местах пришлось бы хранить в двойном массиве (или хеш-таблице, связывающей номер уровня со списком мест). Решение, в котором `ParkingLot` просто отделяется от `Level`, получается более элегантным.

```
1  public class ParkingLot {
2    private Level[] levels;
3    private final int NUM_LEVELS = 5;
4
5    public ParkingLot() { ...   }
6   /* Припарковать транспорт на парковочном месте (или нескольких
7    * местах). Вернуть false в случае неудачи. */
8    public boolean parkVehicle(Vehicle vehicle) { ... }
9  }
10
11  /* Класс представляет уровень парковки */
12  public class Level {
13    private int floor;
14    private ParkingSpot[] spots;
15    private int availableSpots = 0; // Количество свободных мест
16    private static final int SPOTS_PER_ROW = 10;
17
18    public Level(int flr, int numberSpots) { ... }
19
20    public int availableSpots() { return availableSpots; }
21
22    /* Поиск места для парковки. Вернуть false в случае неудачи. */
23    public boolean parkVehicle(Vehicle vehicle) { ...  }
```

```
24
25    /* Парковка машины начинается с места spotNumber и занимает
26     * vehicle.spotsNeeded мест. */
27    private boolean parkStartingAtSpot(int num, Vehicle v) { ...   }
28
29    /* Поиск места для парковки. Вернуть индекс места или -1 при неудаче. */
30    private int findAvailableSpots(Vehicle vehicle) { ... }
31
32    /* При освобождении места увеличить availableSpots */
33    public void spotFreed() { availableSpots++; }
34  }
```

Реализация `ParkingSpot` основана на переменной, описывающей размер парковочного места. Можно реализовать дополнительные классы `LargeSpot`, `CompactSpot` и `MotorcycleSpot`, наследующие от `ParkingSpot`, но это будет лишним. Скорее всего, поведение парковочного места будет оставаться неизменным, различаются только размеры.

```
1  public class ParkingSpot {
2    private Vehicle vehicle;
3    private VehicleSize spotSize;
4    private int row;
5    private int spotNumber;
6    private Level level;
7
8    public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}
9
10    public boolean isAvailable() { return vehicle == null; }
11
12    /* Проверить, что место свободно и имеет достаточный размер. */
13    public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15    /* Парковка на заданном месте. */
16    public boolean park(Vehicle v) { ... }
17
18    public int getRow() { return row; }
19    public int getSpotNumber() { return spotNumber; }
20
21    /* Освобождение места и оповещение уровня о появлении нового места */
22    public void removeVehicle() { ... }
23  }
```

Полная реализация этого решения, включая тестовый код, содержится в архиве примеров.

7.5. Разработайте структуры данных для онлайн-библиотеки.

РЕШЕНИЕ

Поскольку нет точного описания функциональности, давайте предположим, что нам необходимо разработать стандартную онлайн-библиотеку, которая обладает следующей функциональностью:

- Создание и ведение списка подписчиков.
- Поиск книг в базе.

- Чтение.
- В каждый момент времени только один пользователь может быть активен.
- Пользователь может читать только одну книгу.

Для реализации этих операций понадобятся функции `get`, `set`, `update` и т. д. Основными объектами нашей системы будут `User`, `Book` и `Library`.

Класс `OnlineReaderSystem` занимает центральное место в программе. Можно реализовать класс так, что он будет хранить информацию обо всех книгах, управлять пользователями и заниматься обновлением экрана, но в итоге класс окажется слишком большим. Вместо этого лучше разделить функции между классами `Library`, `UserManager` и `Display`.

```
1  public class OnlineReaderSystem {
2    private Library library;
3    private UserManager userManager;
4    private Display display;
5
6    private Book activeBook;
7    private User activeUser;
8
9    public OnlineReaderSystem() {
10     userManager = new UserManager();
11     library = new Library();
12     display = new Display();
13   }
14
15   public Library getLibrary() { return library;  }
16   public UserManager getUserManager() { return userManager;  }
17   public Display getDisplay() { return display; }
18
19   public Book getActiveBook() { return activeBook; }
20   public void setActiveBook(Book book) {
21     activeBook = book;
22     display.displayBook(book);
23   }
24
25   public User getActiveUser() { return activeUser; }
26   public void setActiveUser(User user) {
27     activeUser = user;
28     display.displayUser(user);
29   }
30 }
```

Таким образом, работой с пользователем, библиотекой и выводом текста будут заниматься разные классы:

```
1  public class Library {
2    private HashMap<Integer, Book> books;
3
4    public Book addBook(int id, String details) {
5      if (books.containsKey(id)) {
6        return null;
7      }
8      Book book = new Book(id, details);
9      books.put(id, book);
```

```
10      return book;
11    }
12
13    public boolean remove(Book b) { return remove(b.getID());  }
14    public boolean remove(int id) {
15      if (!books.containsKey(id)) {
16        return false;
17     }
18      books.remove(id);
19      return true;
20    }
21
22    public Book find(int id) {
23      return books.get(id);
24    }
25  }
26
27  public class UserManager {
28    private HashMap<Integer, User> users;
29
30    public User addUser(int id, String details, int accountType) {
31      if (users.containsKey(id)) {
32        return null;
33     }
34      User user = new User(id, details, accountType);
35      users.put(id, user);
36      return user;
37    }
38
39    public User find(int id) { return users.get(id); }
40    public boolean remove(User u) { return remove(u.getID());  }
41    public boolean remove(int id) {
42      if (!users.containsKey(id)) {
43        return false;
44     }
45      users.remove(id);
46      return true;
47    }
48  }
49
50  public class Display {
51    private Book activeBook;
52    private User activeUser;
53    private int pageNumber = 0;
54
55    public void displayUser(User user) {
56      activeUser = user;
57      refreshUsername();
58    }
59
60    public void displayBook(Book book) {
61      pageNumber = 0;
62      activeBook = book;
63
64      refreshTitle();
65      refreshDetails();
66      refreshPage();
```

```
67    }
68
69    public void turnPageForward() {
70      pageNumber++;
71      refreshPage();
72    }
73
74    public void turnPageBackward() {
75      pageNumber--;
76      refreshPage();
77    }
78
79    public void refreshUsername() { /* Обновление имени пользователя */  }
80    public void refreshTitle() { /* Обновление названия */ }
81    public void refreshDetails() { /* Обновление подробной информации */   }
82    public void refreshPage() { /* Обновление страницы */ }
83  }
```

Классы User и Book просто обеспечивают хранение данных и содержат минимум функциональности.

```
1   public class Book {
2     private int bookId;
3     private String details;
4
5     public Book(int id, String det) {
6       bookId = id;
7       details = det;
8     }
9
10    public int getID() { return bookId;   }
11    public void setID(int id) { bookId = id; }
12    public String getDetails() { return details; }
13    public void setDetails(String d) { details = d; }
14  }
15
16  public class User {
17    private int userId;
18    private String details;
19    private int accountType;
20
21    public void renewMembership() {  }
22
23    public User(int id, String details, int accountType) {
24      userId = id;
25      this.details = details;
26      this.accountType = accountType;
27    }
28
29    /* Get- и set-методы */
30    public int getID() { return userId; }
31    public void setID(int id) { userId = id; }
32    public String getDetails() {
33      return details;
34    }
35
36    public void setDetails(String details) {
```

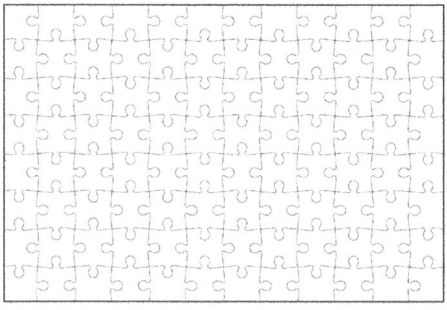
```
37      this.details = details;
38    }
39    public int getAccountType() { return accountType; }
40    public void setAccountType(int t) { accountType = t; }
41  }
```

Решение вынести управление пользователями, библиотекой и экраном в отдельные классы из класса `Onlinereadersystem` довольно интересно. При разработке небольшой системы оно может привести к чрезмерному усложнению кода, но если система будет расширяться, а `onlinereadersystem` будет дополняться новой функциональностью, выделение таких компонентов предотвратит чрезмерное разрастание основного класса.

7.6. Запрограммируйте модель сборной головоломки («пазл») из $N \times N$ фрагментов. Разработайте структуры данных и объясните алгоритм, позволяющий решить задачу. Предполагается, что существует метод `fitsWith`, возвращающий значение `true` в том случае, если два переданных фрагмента головоломки должны располагаться рядом.

РЕШЕНИЕ

Традиционная простая сборная головоломка имеет табличную структуру со строками и столбцами. Каждый фрагмент находится на пересечении конкретной строки и столбца и имеет четыре стороны. Каждая сторона может относиться к одному из трех типов: вогнутая, выпуклая и плоская. Например, угловой фрагмент имеет две плоских стороны и две стороны другого типа, которые могут быть как вогнутыми, так и выпуклыми.



В процессе сбора головоломки (вручную или алгоритмически) необходимо где-то хранить позицию каждого фрагмента. Позиция может быть абсолютной или относительной:

- Абсолютная позиция: «Фрагмент находится на позиции с координатами (12, 23)».
- Относительная позиция: «Я не знаю, где расположен этот фрагмент, но я знаю, что он находится рядом с другим».

Для построения решения мы будем использовать только абсолютную позицию.

Нам понадобятся классы для представления головоломки (`Puzzle`), фрагмента (`Piece`) и стороны (`Edge`). Также понадобятся перечисления для видов сторон

(выпуклая, вогнутая, плоская) и вариантов их ориентации (левая, правая, верхняя, нижняя).

Экземпляр `Puzzle` в исходном состоянии представляет собой список фрагментов. Когда головоломка будет решена, матрица фрагментов `N×N` будет заполнена.

Класс `Piece` содержит хеш-таблицу, связывающую ориентацию с соответствующей стороной. Так как в какой-то момент фрагмент может быть повернут, хеш-таблица может измениться. Исходные ориентации сторон будут назначены случайным образом.

Класс `Edge` содержит только вид стороны и указатель на родительский фрагмент. Информация об ориентации в нем не сохраняется.

Одна из возможных объектно-ориентированных архитектур будет иметь следующий вид:

```
1  public enum Orientation {
2    LEFT, TOP, RIGHT, BOTTOM; // Должны оставаться в таком порядке
3
4    public Orientation getOpposite() {
5      switch (this) {
6        case LEFT: return RIGHT;
7        case RIGHT: return LEFT;
8        case TOP: return BOTTOM;
9        case BOTTOM: return TOP;
10        default: return null;
11     }
12    }
13  }
14
15  public enum Shape {
16    INNER, OUTER, FLAT;
17
18    public Shape getOpposite() {
19      switch (this) {
20        case INNER: return OUTER;
21        case OUTER: return INNER;
22        default: return null;
23     }
24    }
25  }
26
27  public class Puzzle {
28    private LinkedList<Piece> pieces; /* Неразложенные фрагменты. */
29    private Piece[][] solution;
30    private int size;
31
32    public Puzzle(int size, LinkedList<Piece> pieces) { ... }
33
34
35    /* Положить фрагмент, повернуть его и удалить из списка. */
36    private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
37                                   int column, Orientation orientation) {
38      Piece piece = edge.getParentPiece();
39      piece.setEdgeAsOrientation(edge, orientation);
40      pieces.remove(piece);
41      solution[row][column] = piece;
```

```
42    }
43
44    /* Найти фрагмент в piecesToSearch и вставить в позицию row, col. */
45    private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int
col);
46
47    /* Решение головоломки. */
48    public boolean solve() { ...  }
49  }
50
51  public class Piece {
52    private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();
53
54    public Piece(Edge[] edgeList) { ... }
55
56    /* Поворот сторон на "numberRotations". */
57    public void rotateEdgesBy(int numberRotations) { ... }
58
59    public boolean isCorner() { ... }
60    public boolean isBorder() { ... }
61  }
62
63  public class Edge {
64    private Shape shape;
65    private Piece parentPiece;
66    public Edge(Shape shape) { ... }
67    public boolean fitsWith(Edge edge) { ... }
68  }
```

Алгоритм решения головоломки

Будем действовать так же, как обычно при решении головоломок. Начнем с группировки фрагментов на угловые, боковые и внутренние.

Когда это будет сделано, выберем произвольный угловой фрагмент и поместим его в левый верхний угол. Затем программа последовательно перебирает головоломку и заполняет ее фрагмент за фрагментом. Для каждой позиции она ищет подходящий фрагмент в соответствующей группе. При вставке в головоломку фрагмент разворачивается в нужной ориентации.

Следующий код описывает общую структуру алгоритма.

```
1  /* Поиск фрагмента в piecesToSearch и вставка его в позицию row, col. */
2  boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
3    if (row == 0 && column == 0) { // Левый верхний угол - просто вставить
4      Piece p = piecesToSearch.remove();
5      orientTopLeftCorner(p);
6      solution[0][0] = p;
7    } else {
8      /* Определение подходящей стороны. */
9      Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
10                                        solution[row][column - 1];
11     Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
12                                                   Orientation.RIGHT;
13     Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);
14
15     /* Поиск подходящей стороны. */
```

```
    Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
    if (edge == null) return false; // Решение невозможно

    /* Вставка фрагмента и включение стороны. */
    Orientation orientation = orientationToMatch.getOpposite();
    setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
  }
  return true;
}

boolean solve() {
  /* Группировка фрагментов. */
  LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
  LinkedList<Piece> borderPieces = new LinkedList<Piece>();
  LinkedList<Piece> insidePieces = new LinkedList<Piece>();
  groupPieces(cornerPieces, borderPieces, insidePieces);

  /* Перебор с поиском фрагмента, стыкующегося с предыдущим. */
  solution = new Piece[size][size];
  for (int row = 0; row < size; row++) {
    for (int column = 0; column < size; column++) {
      LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
        borderPieces, insidePieces, row, column);
      if (!fitNextEdge(piecesToSearch, row, column)) {
        return false;
      }
     }
  }

  return true;
}
```

Полный код решения содержится в архиве примеров.

7.7. Как бы вы подошли к проектированию чат-сервера? Предоставьте информацию о компонентах внутренней подсистемы (backend), классах и методах. Перечислите самые трудные задачи, которые необходимо решить.

РЕШЕНИЕ

Разработка чат-сервера — огромный проект, и конечно, вам никак не удастся представить законченное решение в рамках собеседования. В конце концов, команды из многих людей проводят месяцы и годы, пытаясь создать хороший чат-сервер. Вы, как кандидат, должны сосредоточиться на отдельном аспекте задачи, достаточно четко определенном, чтобы его можно было завершить во временных рамках собеседования. Ваше решение не обязано точно отражать реальность, но оно должно давать достаточно хорошее представление о фактической реализации.

Мы займемся основными аспектами управления пользователями и организацией общения: добавление пользователя, создание беседы, обновление статуса и т. д. Для экономии времени и места мы не будем углубляться в сетевые аспекты задачи и не будем рассматривать вопросы передачи данных между клиентами.

Предполагается, что «дружба» взаимна: вы дружите со мной только в том случае, если и я дружу с вами. Наша чат-система будет поддерживать как групповые, так

и приватные беседы. Мы не будем рассматривать голосовой чат, видеочат и передачу файлов.

Какие конкретные операции необходимо поддерживать?

Этот вопрос следует обсудить с интервьюером, но мы приведем несколько примеров:

- Вход и выход из чата.
- Добавление запроса (отправка, прием и отклонение).
- Обновление информации о статусе.
- Создание приватных и групповых чатов.
- Добавление новых сообщений в приватные и групповые чаты.

Этот список далеко не полон. Если вам хватит времени, добавьте дополнительные действия.

Что можно узнать из этих требований?

Необходимо реализовать концепцию пользователей, статуса добавления запроса, статуса подключения и сообщений.

Какие базовые компоненты должны присутствовать в системе?

Система будет состоять из базы данных, клиентов и серверов. Мы не включаем эти части в объектно-ориентированный проект, но можем обсудить общее видение системы.

База данных будет использоваться для долгосрочного хранения данных, например списка пользователей и архивов чата. В большинстве случаев подойдет SQL-база данных, но если нам понадобится большая масштабируемость, можно использовать BigTable или другую аналогичную систему.

Для обмена данными между клиентами и серверами подойдет XML. Хотя его компактность оставляет желать лучшего (вы обязательно должны сказать об этом интервьюеру), он наиболее удобен для восприятия как человеком, так и компьютером. Использование XML также упрощает отладку приложения, а это имеет большое значение.

Сервер будет состоять из множества компьютеров. Данные будут распределены между машинами, что требует «переключения» с одного устройства на другое. Возможно, некоторые данные придется перераспределять между машинами, чтобы сократить время поиска. Нужно избегать узких мест. Например, если аутентификацией пользователей занимается только одна машина, то выход ее из строя закроет доступ к системе миллионам пользователей.

Какие основные объекты и методы должны поддерживаться системой?

Ключевые объекты нашей системы — пользователи, беседы и сообщения о статусах. Все это можно реализовать в классе `UserManager`. Если бы мы уделяли больше внимания сетевым аспектам задачи или другим компонентам, вероятно, нам пришлось бы создать для них дополнительные объекты.

```
   /* UserManager - центральный класс для основных действий пользователя. */
1  public class UserManager {
2    private static UserManager instance;
3    /* Связывает идентификатор с пользователем */
4    private HashMap<Integer, User> usersById;
5
6    /* Связывает имя учетной записи с пользователем */
7    private HashMap<String, User> usersByAccountName;
8
9    /* Связывает идентификатор пользователя с подключенным пользователем */
10    private HashMap<Integer, User> onlineUsers;
11
12    public static UserManager getInstance() {
13      if (instance == null) instance = new UserManager();
14      return instance;
15    }
16
17    public void addUser(User fromUser, String toAccountName) { ... }
18    public void approveAddRequest(AddRequest req) { ...  }
19    public void rejectAddRequest(AddRequest req) { ... }
20    public void userSignedOn(String accountName) { ... }
21    public void userSignedOff(String accountName) { ... }
22 }
```

Метод `receivedAddRequest` класса `User` оповещает пользователя B о том, что пользователь A запросил добавление его в список контактов. Пользователь B соглашается или отклоняет запрос (при помощи `UserManager.approveAddRequest` или `rejectAddRequest`), а класс `UserManager` обеспечивает добавление пользователей в списки контактов двух пользователей.

Метод `sentAddRequest` класса `User` вызывается `UserManager` для добавления `AddRequest` в список запросов пользователя A. Следовательно, последовательность операций должна выглядеть так:

1. Пользователь A нажимает кнопку «добавить пользователя» в клиентской программе, запрос отправляется на сервер.
2. Пользователь B вызывает `requestAddUser(User B)`.
3. Этот метод вызывает `UserManager.addUser`.
4. `UserManager` вызывает методы `UserA.sentAddRequest` и `UserB.receivedAddRequest`.

Это всего лишь *один из возможных* способов проектирования подобных взаимодействий, но не единственный и даже не самый лучший.

```
1  public class User {
2    private int id;
3    private UserStatus status = null;
4
5    /* Связывает идентификатор пользователя другого участника с чатом */
6    private HashMap<Integer, PrivateChat> privateChats;
7
8    /* Список групповых чатов */
9    private ArrayList<GroupChat> groupChats;
10
11    /* Связывает идентификатор другого пользователя с запросом на добавление */
12    private HashMap<Integer, AddRequest> receivedAddRequests;
13
```

```
14    /* Связывает идентификатор другого пользователя с запросом на добавление*/
15    private HashMap<Integer, AddRequest> sentAddRequests;
16
17    /* Связывает идентификатор пользователя с объектом пользователя */
18    private HashMap<Integer, User> contacts;
19
20    private String accountName;
21    private String fullName;
22
23    public User(int id, String accountName, String fullName) { ... }
24    public boolean sendMessageToUser(User to, String content){ ... }
25    public boolean sendMessageToGroupChat(int id, String cnt){...}
26    public void setStatus(UserStatus status) { ... }
27    public UserStatus getStatus() { ... }
28    public boolean addContact(User user) { ... }
29    public void receivedAddRequest(AddRequest req) { ...  }
30    public void sentAddRequest(AddRequest req) { ... }
31    public void removeAddRequest(AddRequest req) { ... }
32    public void requestAddUser(String accountName) { ... }
33    public void addConversation(PrivateChat conversation) { ... }
34    public void addConversation(GroupChat conversation) { ... }
35    public int getId() { ... }
36    public String getAccountName() { ... }
37    public String getFullName() { ... }
38  }
```

Класс `Conversation` реализован как абстрактный, потому что все экземпляры `Conversation` должны относиться либо к классу `GroupChat`, либо к классу `PrivateChat`, а каждый из этих классов обладает собственной функциональностью.

```
1  public abstract class Conversation {
2    protected ArrayList<User> participants;
3    protected int id;
4    protected ArrayList<Message> messages;
5
6    public ArrayList<Message> getMessages() { ... }
7    public boolean addMessage(Message m) { ... }
8    public int getId() { ... }
9  }
10
11 public class GroupChat extends Conversation {
12   public void removeParticipant(User user) { ... }
13   public void addParticipant(User user) { ... }
14 }
15
16 public class PrivateChat extends Conversation {
17   public PrivateChat(User user1, User user2) { ...
18   public User getOtherParticipant(User primary) { ... }
19 }
20
21 public class Message {
22   private String content;
23   private Date date;
24   public Message(String content, Date date) { ... }
25   public String getContent() { ... }
26   public Date getDate() { ... }
27 }
```

Классы `AddRequest` и `UserStatus` — простые классы с минимальной функциональностью. Их основное назначение — группировка данных, используемых другими классами.

```
1  public class AddRequest {
2    private User fromUser;
3    private User toUser;
4    private Date date;
5    RequestStatus status;
6
7    public AddRequest(User from, User to, Date date) { ... }
8    public RequestStatus getStatus() { ... }
9    public User getFromUser() { ... }
10    public User getToUser() { ... }
11    public Date getDate() { ... }
12 }
13
14 public class UserStatus {
15    private String message;
16    private UserStatusType type;
17    public UserStatus(UserStatusType type, String message) { ... }
18    public UserStatusType getStatusType() { ... }
19    public String getMessage() { ... }
20 }
21
22 public enum UserStatusType {
23   Offline, Away, Idle, Available, Busy
24 }
25
26 public enum RequestStatus {
27   Unread, Read, Accepted, Rejected
28 }
```

В архиве примеров содержится более подробная версия этого кода, включая реализацию методов, приведенных выше.

Какие проблемы окажутся самыми трудными (или интересными)?

Приведенные далее вопросы стоит обсудить с интервьюером.

Вопрос 1. Как узнать (с полной уверенностью), кто подключен к системе в настоящий момент?

Хотелось бы, чтобы пользователи явно сообщали о завершении работы с системой, но полной уверенности в этом быть не может. Например, подключение пользователя может «зависнуть». Чтобы убедиться в том, что пользователь продолжает работать с системой, можно проводить регулярный опрос клиента.

Вопрос 2. Что делать с информационными конфликтами?

Часть информации должна храниться в оперативной памяти, а часть — в базе данных. Что случится, если синхронизация между ними будет нарушена? Какую информацию считать правильной?

Вопрос 3. Что делать с масштабируемостью сервера?

В ходе проектирования чат-сервера мы не особенно беспокоились о масштабируемости, но в реальной ситуации этой проблемой придется заниматься. Данные

необходимо распределить по нескольким серверам и реализовать механизм синхронизации данных.

Вопрос 4. Как защититься от DoS-атак?

Клиенты могут передавать нам данные, а что, если нас попытаются «завалить» потоком запросов? Как предотвратить атаку?

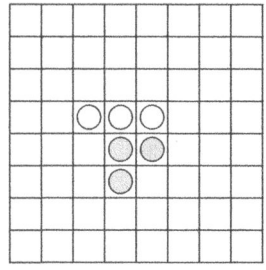
7.8. В «реверси» играют по следующим правилам: каждая фишка в игре с одной стороны белая, а с другой — черная. Когда ряд фишек оказывается ограничен фишками противника (слева и справа или сверху и снизу), цвет фишек в этом ряду меняется на противоположный. При своем ходе игрок обязан захватить по крайней мере одну из фишек противника, если это возможно. Игра заканчивается, когда у обоих игроков не остается допустимых ходов. Побеждает тот, у которого больше фишек на поле. Реализуйте ООП-модель для этой игры.

РЕШЕНИЕ

Давайте начнем с примера. Предположим, что в ходе игры сделаны следующие ходы:

1. Начальное состояние доски — две черные и две белые фишки в центре (черные фишки стоят в верхнем левом и в нижнем правом углах центрального квадрата).
2. Ход черных `(6, 4)` меняет цвет фишки `(5, 4)` с белого на черный.
3. Ход белых `(4, 3)` меняет цвет фишки `(4, 4)` с черного на белый.

После данной последовательности ходов доска имеет следующий вид:



Вероятно, основные объекты должны представлять текущую партию, доску, фишки (черные или белые) и игроков. Как представить все это в объектно-ориентированной архитектуре?

Нужны ли классы BlackPiece и WhitePiece?

При первом взгляде на задачу кажется, что нам понадобятся два класса, `BlackPiece` и `WhitePiece`, которые представляют фишки и наследуют от абстрактного класса `Piece`. Однако это не самая лучшая идея. Каждая фишка может менять цвет, следовательно, нам придется постоянно уничтожать и создавать экземпляры, которые, по сути, представляют один объект, а это не очень разумно. Вероятно, будет проще создать класс `Piece` с флагом, соответствующим текущему цвету.

Нужны ли отдельные классы Board и Game?

Строго говоря, нам не нужны отдельные объекты `Board` и `Game`. Однако разделение объектов позволяет организовать логическое разделение функциональных задач между доской (логика перемещения фишек) и игрой (время игры, ход игры и т. д.). В такой системе есть небольшой недостаток: в программе появляются дополнительные логические уровни. Функция может вызвать метод `Game` только для того, чтобы он немедленно вызвал метод `Board`. Мы разделим объекты `Game` и `Board`, но на собеседовании вам следует обсудить этот момент с интервьюером.

Как хранить результат?

Вероятно, нужно хранить количество черных и белых фишек. Но кто должен поддерживать эту информацию? Было бы логично возложить эту задачу на `Game` или `Board` или даже на `Piece` (в статических методах). Мы будем хранить эту информацию в `Board`, потому что она логически группируется с состоянием доски. Она обновляется объектами `Piece` или `Board` при вызове методов `colorChanged` и `colorAdded` класса `Board`.

Должен ли Game быть синглетным классом?

Реализация `Game` в виде синглетного класса имеет определенное преимущество: она позволяет легко вызвать метод из `Game` без передачи ссылки на объект `Game`.

Но это означает, что в программе будет существовать только один экземпляр `Game`. Можно ли сделать такое предположение? Лучше обсудить этот нюанс с интервьюером.

Одна из возможных реализаций «Реверси» выглядит так:

```
1  public enum Direction {
2    left, right, up, down
3  }
4
5  public enum Color {
6    White, Black
7  }
8
9  public class Game {
10   private Player[] players;
11   private static Game instance;
12   private Board board;
13   private final int ROWS = 10;
14   private final int COLUMNS = 10;
15
16   private Game() {
17     board = new Board(ROWS, COLUMNS);
18     players = new Player[2];
19     players[0] = new Player(Color.Black);
20     players[1] = new Player(Color.White);
21   }
22
23   public static Game getInstance() {
24     if (instance == null) instance = new Game();
25     return instance;
```

```
26    }
27
28    public Board getBoard() {
29      return board;
30    }
31  }
```

Класс `Board` управляет фишками. Он не управляет процессом игры, оставляя эту задачу для класса `Game`.

```
1  public class Board {
2    private int blackCount = 0;
3    private int whiteCount = 0;
4    private Piece[][] board;
5
6    public Board(int rows, int columns) {
7      board = new Piece[rows][columns];
8    }
9
10    public void initialize() {
11      /* Инициализировать начальную расстановку фишек */
12    }
13
14    /* Попытаться поместить фишку цвета color в позицию (row, column).
15     * Вернуть true, если попытка оказалась успешной. */
16    public boolean placeColor(int row, int column, Color color) {
17      ...
18    }
19
20    /* Поменять цвет фишек от позиции (row, column) в направлении d. */
21    private int flipSection(int row, int column, Color color, Direction d) { ...
}
22
23    public int getScoreForColor(Color c) {
24      if (c == Color.Black) return blackCount;
25      else return whiteCount;
26    }
27
28    /* Обновить доску дополнительными фишками newPieces цвета newColor.
29     * Уменьшить количество фишек противоположного цвета. */
30    public void updateScore(Color newColor, int newPieces) { ... }
31  }
```

Как упоминалось ранее, для представления черных и белых фишек используется класс `Piece` с простой переменной `Color`, представляющей цвет фишки (черная или белая).

```
1  public class Piece {
2    private Color color;
3    public Piece(Color c) { color = c; }
4
5    public void flip() {
6      if (color == Color.Black) color = Color.White;
7      else color = Color.Black;
8    }
9
10    public Color getColor() { return color; }
11  }
```

Класс `Player` содержит минимум информации. В нем даже не хранится текущий счет, хотя и есть метод, позволяющий получить счет конкретного игрока, — `getScore()`. Для получения нужной информации `Player.getScore()` обращается с вызовом к объекту `Game`.

```
1  public class Player {
2    private Color color;
3    public Player(Color c) { color = c; }
4
5    public int getScore() { ...   }
6
7    public boolean playPiece(int r, int c) {
8      return Game.getInstance().getBoard().placeColor(r, c, color);
9    }
10
11    public Color getColor() { return color; }
12 }
```

Полностью функциональную (автоматизированную) версию этого кода можно найти в архиве примеров.

Помните, что во многих задачах важно не то, что вы сделали, а то, *почему* вы это сделали. Интервьюеру все равно, реализовали вы класс `Game` в синглетной форме или нет, но ему будет важно, чтобы вы задумались над происходящим и обсудили достоинства и недостатки разных решений.

7.9. Реализуйте класс `CircularArray` для представления структуры данных — аналога массива с эффективной реализацией циклического сдвига. Если возможно, класс должен использовать обобщенный (generic) тип и поддерживать перебор в стандартном синтаксисе (`Obj o : circularArray`).

РЕШЕНИЕ

Задача на самом деле состоит из двух частей: сначала необходимо реализовать класс `CircularArray`, а затем обеспечить поддержку перебора элементов. Рассмотрим обе части по отдельности.

Реализация класса CircularArray

Один из вариантов реализации класса `CircularArray` — непосредственное выполнение циклического сдвига элементов при вызове `rotate(int shiftRight)`. Конечно, этот способ нельзя назвать эффективным.

Вместо этого можно создать переменную экземпляра `head`, указывающую на *концептуальное* начало циклического массива. Вместо того чтобы сдвигать элементы в массиве, мы просто увеличим значение `head` на `shiftRight`.

Реализация этого метода приведена ниже:

```
1  public class CircularArray<T> {
2    private T[] items;
3    private int head = 0;
4
5    public CircularArray(int size) {
6      items = (T[]) new Object[size];
```

```
7    }
8
9    private int convert(int index) {
10      if (index < 0) {
11        index += items.length;
12     }
13      return (head + index) % items.length;
14    }
15
16    public void rotate(int shiftRight) {
17      head = convert(shiftRight);
18    }
19
20    public T get(int i) {
21      if (i < 0 || i >= items.length) {
22        throw new java.lang.IndexOutOfBoundsException("...");
23     }
24      return items[convert(i)];
25    }
26
27    public void set(int i, T item) {
28      items[convert(i)] = item;
29    }
30  }
```

В этом коде есть несколько неочевидных мест, в которых легко ошибиться:

- В Java нельзя создать массив обобщенного типа. Вместо этого необходимо либо выполнить преобразование типа, либо определить `items` с типом `List<T>`. Для простоты мы выбрали первый вариант.
- Оператор `%` будет возвращать отрицательное значение в выражении `negValue % posVal` (например, `-8 % 3 = -2`). Это не соответствует математическому определению операции деления с остатком. Чтобы получить правильный положительный результат, к отрицательному значению индекса необходимо прибавить `items.length`.
- Необходимо следить за тем, чтобы необработанный индекс всегда преобразовывался в сдвинутый. Для этого в код добавлена функция `convert`, которая используется другими методами. Даже в функции `rotate` используется `convert`. Это хороший пример повторного использования кода.

Теперь у нас есть основной код `CircularArray`, и мы можем сосредоточиться на реализации итератора.

Реализация интерфейса Iterator

Вторая часть задачи — реализовать класс `CircularArray` таким образом, чтобы он позволял выполнять следующую операцию:

```
1  CircularArray<String> array = ...
2  for (String s : array) { ... }
```

Для этого необходимо реализовать интерфейс `Iterator`. Подробности реализации относятся к специфике Java, но аналогичные возможности могут быть реализованы и в других языках.

Чтобы реализовать интерфейс `Iterator`, мы должны сделать следующее:

- изменить определение `CircularArray<T>` и добавить `implements Iterable<T>`; также потребуется добавить метод `iterator()` в `CircularArray<T>`;
- создать класс `CircularArrayIterator()`, реализующий `Iterator<T>`. Для этого также необходимо реализовать в `CircularArrayIterator` методы `hasNext()`, `next()` и `remove()`.

Как только это будет сделано, цикл `for` заработает.

В приведенном далее коде мы опустили те аспекты `CircularArray`, которые идентичны предыдущей реализации:

```
public class CircularArray<T> implements Iterable<T> {
  ...
  public Iterator<T> iterator() {
    return new CircularArrayIterator<T>(this);
  }

  private class CircularArrayIterator<TI> implements Iterator<TI>{
    /* Значение current отражает смещение от "повернутого" начала,
     * а не от фактического начала базового массива. */
     private int _current = -1;
     private TI[] _items;

     public CircularArrayIterator(CircularArray<TI> array){
       _items = array.items;
     }

     @Override
     public boolean hasNext() {
       return _current < items.length - 1;
     }

     @Override
     public TI next() {
       _current++;
       TI item = (TI) _items[convert(_current)];
       return item;
     }

     @Override
     public void remove() {
       throw new UnsupportedOperationException("...");
     }
   }
}
```

Обратите внимание: первая итерация цикла `for` вызывает `hasNext()`, а затем `next()`. Будьте внимательны и следите за тем, чтобы ваша реализация возвращала правильные значения.

Если вам на собеседовании достанется подобная задача, возможно, вы не вспомните точные названия методов и интерфейсов. В этом случае попытайтесь решить задачу так, как можете. Если вы сможете объяснить, какие методы вам для этого понадобятся, уже одно это продемонстрирует вашу квалификацию.

7.10. Спроектируйте и реализуйте текстовый вариант игры «Сапер». В этой классической компьютерной игре для одного игрока на поле $N \times N$ тайно расставляются B мин. Остальные ячейки либо пусты, либо в них выводится цифра — количество мин в окружающих восьми ячейках. Игрок открывает ячейку. Если в ней окажется мина, то игрок проигрывает. Если ячейка содержит число, то игрок видит это число. Если ячейка пуста, то открывается эта ячейка со всеми прилегающими пустыми ячейками (до окружающих ячеек с цифрами). Игрок побеждает, если ему удастся открыть все ячейки, не содержащие мин. Также игрок может помечать некоторые ячейки как потенциальные мины. На ход игры такая пометка не влияет, она лишь предотвращает случайные щелчки на ячейках, в которых, по мнению игрока, находится мина. (Подсказка: если эта игра вам незнакома, сыграйте несколько партий в Интернете.)

Игровое поле с тремя минами. Изначально игрок не видит его.

	1	1	1			
	1	*	1			
	2	2	2			
	1	*	1			
	1	1	1			
			1	1	1	
			1	*	1	

В начале игры выводится полностью закрытое поле.

?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

Щелчок на ячейке (строка=1, столбец=0) открывает часть игрового поля:

	1	?	?	?	?	?
	1	?	?	?	?	?
	2	?	?	?	?	?
	1	?	?	?	?	?
	1	1	1	?	?	?
			1	?	?	?
			1	?	?	?

Пользователь выигрывает, если открыты все ячейки, не содержащие мин.

	1	1	1			
	1	?	1			
	2	2	2			
	1	?	1			
	1	1	1			
			1	1	1	
			1	?	1	

РЕШЕНИЕ

Написание полной игры — даже текстовой — потребует значительно большего времени, чем то, что выделено для собеседования. Впрочем, это вовсе не означает, что вопрос нечестный. Просто интервьюер не ждет, что вы напишете весь код во время собеседования. Кроме того, предполагается, что вы сосредоточитесь на проработке ключевых идей или структуры.

Начнем с классов. Безусловно, понадобятся классы для представления ячейки (`Cell`) и для представления игрового поля (`Board`). Вероятно, также стоит иметь класс для представления партии (`Game`).

Теоретически классы Board и Game можно было бы объединить, но лучше держать их по отдельности. Избыточное структурирование лучше недостаточного. Класс Board может хранить список объектов Cell и выполнять базовые ходы с переворачиванием фишек. Класс Game хранит текущее состояние партии и обрабатывает данные, введенные пользователем.

Классу Cell необходимо знать, что находится в текущей ячейке — мина, число или ничего. Для хранения этой информации можно было бы создать субкласс Cell, но вряд ли это принесет какую-нибудь пользу.

Также для описания типа ячейки можно было бы ввести перечисление enum TYPE {BOMB, NUMBER, BLANK}. Мы не стали этого делать, потому что BLANK в действительности является разновидностью NUMBER с нулевым значением. Достаточно просто добавить флаг isBomb.

Если вы приняли другие решения — это нормально. Предложенная архитектура не является лучшей из всех возможных. Объясните принятые решения, а также их достоинства и недостатки своему интервьюеру.

Также необходимо хранить информацию о том, открыта ячейка или нет. Вероятно, создавать на базе Cell субклассы ExposedCell и UnexposedCell было бы неразумно, ведь в Board хранится ссылка на ячейки, которая должна изменяться при изменении состояния ячейки. Но что произойдет, если ссылка на экземпляр Cell также хранится в других объектах?

Лучше ввести флаг isExposed — признак открытой ячейки. Аналогичным образом мы поступим для isGuess.

```
1  public class Cell {
2    private int row;
3    private int column;
4    private boolean isBomb;
5    private int number;
6    private boolean isExposed = false;
7    private boolean isGuess = false;
8
9    public Cell(int r, int c) { ... }
10
11   /* Get- и set-методы для перечисленных выше переменных. */
12   ...
13
14   public boolean flip() {
15     isExposed = true;
16     return !isBomb;
17   }
18
19   public boolean toggleGuess() {
20     if (!isExposed) {
21       isGuess = !isGuess;
22    }
23     return isGuess;
24   }
25
26   /* Полный код приведен в архиве примеров. */
27 }
```

Проектирование: Board

В классе `Board` должен храниться массив всех объектов `Cell`. Двумерный массив вполне подойдет.

Вероятно, в объекте `Board` также должна храниться информация о количестве неоткрытых ячеек. Мы будем отслеживать это значение, чтобы его не приходилось вычислять снова и снова.

В классе `Board` также будут реализованы некоторые базовые алгоритмы:

- Инициализация игрового поля и расстановка мин.
- Изменение состояния ячейки.
- Расширение пустых областей.

Класс получает ход от объекта `Game` и реализует его. Затем он должен вернуть результат хода — ячейка с миной (проигрыш), ход за пределами поля, ход в уже открытой области, пустая ячейка (продолжение игры), пустая ячейка (выигрыш), ячейка с цифрой (выигрыш). Вернуть необходимо два элемента данных: признак успеха (был ли ход сделан успешно) и состояние игры (выигрыш, проигрыш, продолжение игры). Для возвращения данных используется дополнительный класс `GamePlayResult`.

Класс `GamePlay` используется для хранения хода, сделанного игроком. В нем хранится строка, столбец и флаг, указывающий, была ли открыта новая информация или пользователь просто пометил ячейку как потенциальную мину.

Основной код класса выглядит примерно так:

```
1  public class Board {
2    private int nRows;
3    private int nColumns;
4    private int nBombs = 0;
5    private Cell[][] cells;
6    private Cell[] bombs;
7    private int numUnexposedRemaining;
8
9    public Board(int r, int c, int b) { ... }
10
11   private void initializeBoard() { ... }
12   private boolean flipCell(Cell cell) { ... }
13   public void expandBlank(Cell cell) { ... }
14   public UserPlayResult playFlip(UserPlay play) { ... }
15   public int getNumRemaining() { return numUnexposedRemaining; }
16 }
17
18 public class UserPlay {
19   private int row;
20   private int column;
21   private boolean isGuess;
22   /* Конструктор, get- и set-методы. */
23 }
24
25 public class UserPlayResult {
26   private boolean successful;
27   private Game.GameState resultingState;
28   /* Конструктор, get- и set-методы. */
29 }
```

Проектирование: Game

В классе `Game` хранится ссылка на игровое поле и данные текущего состояния игры. Также класс получает пользовательский ввод и передает его `Board`.

```
1  public class Game {
2    public enum GameState { WON, LOST, RUNNING }
3
4    private Board board;
5    private int rows;
6    private int columns;
7    private int bombs;
8    private GameState state;
9
10    public Game(int r, int c, int b) { ... }
11
12    public boolean initialize() { ... }
13    public boolean start() { ... }
14    private boolean playGame() { ... } // Цикл до окончания игры.
15 }
```

Алгоритмы

Так выглядит основная объектно-ориентированная архитектура нашего кода. Интервьюер может спросить, как вы реализуете некоторые наиболее интересные алгоритмы.

В данном случае наибольший интерес представляют три алгоритма: инициализация (случайная расстановка мин), присваивание значений нумерованных ячеек и расширение пустой области.

Размещение мин

Простейший алгоритм расстановки мин — случайно выбрать ячейку и поставить мину, если ячейка свободна. Если же ячейка занята, выбирается другое место. Проблема в том, что при большом количестве мин процесс расстановки может стать очень медленным. Возникает опасность многократного выбора ячеек с минами.

Для решения проблемы можно воспользоваться методом, сходным с задачей о тасовании колоды карт (с. 190): расставить `K` мин в первых `K` ячейках, а затем сгенерировать случайную перестановку всех ячеек.

Случайная перестановка генерируется перебором массива от `i`=0 до `N`-1. Для каждого `i` выбирается случайный индекс в диапазоне от `i`=0 до `N`-1, и элементы меняются местами.

Перестановка на игровом поле осуществляется практически так же, просто индекс заменяется позицией из строки и столбца.

```
1  void shuffleBoard() {
2    int nCells = nRows * nColumns;
3    Random random = new Random();
4    for (int index1 = 0; index1 < nCells; index1++) {
5      int index2 = index1 + random.nextInt(nCells - index1);
6      if (index1 != index2) {
7        /* Получение ячейки в позиции index1. */
8        int row1 = index1 / nColumns;
```

```
9        int column1 = (index1 - row1 * nColumns) % nColumns;
10        Cell cell1 = cells[row1][column1];
11
12        /* Получение ячейки в позиции index2. */
13        int row2 = index2 / nColumns;
14        int column2 = (index2 - row2 * nColumns) % nColumns;
15        Cell cell2 = cells[row2][column2];
16
17        /* Перестановка. */
18        cells[row1][column1] = cell2;
19        cell2.setRowAndColumn(row1, column1);
20        cells[row2][column2] = cell1;
21        cell1.setRowAndColumn(row2, column2);
22     }
23    }
24  }
```

Присваивание значений нумерованных ячеек

После того как мины будут расставлены, необходимо присвоить значения нумерованным ячейкам. Можно перебрать все ячейки и проверить, сколько мин окружает каждую. Такое решение работает, но медленнее, чем это реально необходимо.

Вместо этого можно перебрать все мины и увеличить значение в каждой окружающей ее ячейке. Например, если рядом с ячейкой находятся 3 мины, метод `incrementNumber` будет вызван три раза, и в итоге ячейке будет присвоено значение 3.

```
1  /* Заполнение числами ячеек рядом с минами. Хотя мины расставлены
2   * случайно, ссылка на массив bombs указывает на тот же объект. */
3  void setNumberedCells() {
4    int[][] deltas = { // Offsets of 8 surrounding cells
5        {-1, -1}, {-1, 0}, {-1, 1},
6        { 0, -1},          { 0, 1},
7        { 1, -1}, { 1, 0}, { 1, 1}
8    };
9    for (Cell bomb : bombs) {
10      int row = bomb.getRow();
11      int col = bomb.getColumn();
12      for (int[] delta : deltas) {
13        int r = row + delta[0];
14        int c = col + delta[1];
15        if (inBounds(r, c)) {
16          cells[r][c].incrementNumber();
17      }
18     }
19    }
20  }
```

Расширение пустой области

Расширение пустых областей может быть реализовано как итеративно, так и рекурсивно. В нашем примере использована итеративная реализация.

Алгоритм можно описать следующим образом: рядом с каждой пустой ячейкой находятся либо пустые ячейки, либо нумерованные (но никогда ячейки с бомбами). Все они должны быть открыты. Но при открытии пустой ячейки также необходимо добавить ее в очередь, чтобы открыть соседние с ней ячейки.

```
1  void expandBlank(Cell cell) {
2    int[][] deltas = {
3        {-1, -1}, {-1, 0}, {-1, 1},
4        { 0, -1},          { 0, 1},
5        { 1, -1}, { 1, 0}, { 1, 1}
6    };
7
8    Queue<Cell> toExplore = new LinkedList<Cell>();
9    toExplore.add(cell);
10
11    while (!toExplore.isEmpty()) {
12      Cell current = toExplore.remove();
13
14      for (int[] delta : deltas) {
15        int r = current.getRow() + delta[0];
16        int c = current.getColumn() + delta[1];
17
18        if (inBounds(r, c)) {
19          Cell neighbor = cells[r][c];
20          if (flipCell(neighbor) && neighbor.isBlank()) {
21            toExplore.add(neighbor);
22       }
23      }
24     }
25    }
26  }
```

Также алгоритм может быть реализован рекурсивно. В этом случае вместо того, чтобы добавлять ячейку в очередь, следует сделать рекурсивный вызов.

Ваша реализация алгоритмов может существенно различаться в зависимости от выбранной иерархии классов.

7.11. Объясните, какие структуры данных и алгоритмы вы бы использовали для разработки файловой системы, хранящейся в оперативной памяти. Напишите программный код, иллюстрирующий использование этих алгоритмов.

РЕШЕНИЕ

Многие кандидаты, сталкиваясь с подобной задачей, начинают паниковать. Ведь файловая система относится к низкоуровневым задачам!

Не паникуйте! Если правильно выбрать компоненты файловой системы, то можно превратить эту задачу в задачу ООП.

Файловая система в упрощенном представлении состоит из файлов (`File`) и каталогов (`Directory`). Каждый из каталогов в свою очередь может содержать множество других файлов и каталогов. У файла и каталога есть много общих характеристик, поэтому они наследуются от одного и того же класса `Entry`.

```
1  public abstract class Entry {
2    protected Directory parent;
3    protected long created;
4    protected long lastUpdated;
5    protected long lastAccessed;
6    protected String name;
7
```

```
8    public Entry(String n, Directory p) {
9      name = n;
10     parent = p;
11     created = System.currentTimeMillis();
12     lastUpdated = System.currentTimeMillis();
13     lastAccessed = System.currentTimeMillis();
14   }
15
16   public boolean delete() {
17     if (parent == null) return false;
18     return parent.deleteEntry(this);
19   }
20
21   public abstract int size();
22
23   public String getFullPath() {
24     if (parent == null) return name;
25     else return parent.getFullPath() + "/" + name;
26   }
27
28   /* Get- и set-методы. */
29   public long getCreationTime() { return created; }
30   public long getLastUpdatedTime() { return lastUpdated; }
31   public long getLastAccessedTime() { return lastAccessed; }
32   public void changeName(String n) { name = n; }
33   public String getName() { return name; }
34 }
35
36 public class File extends Entry {
37   private String content;
38   private int size;
39
40   public File(String n, Directory p, int sz) {
41     super(n, p);
42     size = sz;
43   }
44
45   public int size() { return size; }
46   public String getContents() { return content; }
47   public void setContents(String c) { content = c; }
48 }
49
50 public class Directory extends Entry {
51   protected ArrayList<Entry> contents;
52
53   public Directory(String n, Directory p) {
54     super(n, p);
55     contents = new ArrayList<Entry>();
56   }
57
58   public int size() {
59     int size = 0;
60     for (Entry e : contents) {
61       size += e.size();
62     }
63     return size;
64   }
65
```

```
66    public int numberOfFiles() {
67      int count = 0;
68      for (Entry e : contents) {
69        if (e instanceof Directory) {
70          count++; // Каталог является разновидностью файла
71          Directory d = (Directory) e;
72          count += d.numberOfFiles();
73        } else if (e instanceof File) {
74          count++;
75      }
76     }
77      return count;
78    }
79
80    public boolean deleteEntry(Entry entry) {
81      return contents.remove(entry);
82    }
83
84    public void addEntry(Entry entry) {
85      contents.add(entry);
86    }
87
88    protected ArrayList<Entry> getContents() { return contents; }
89  }
```

Также можно было реализовать класс `Directory` так, чтобы в нем велись отдельные списки для файлов и подкаталогов. Такая реализация несколько упрощает метод `numberOfFiles()`, в котором не нужно использовать оператор `instanceof`, но усложняет сортировку файлов и каталогов по датам и именам.

7.12. Спроектируйте и реализуйте хеш-таблицу, использующую связные списки для обработки коллизий.

РЕШЕНИЕ

Предположим, вы реализуете хеш-таблицу `Hash<K, V>`, которая связывает объекты типа `K` (ключи) с объектами типа `V` (значениями).

На первый взгляд структура данных могла бы выглядеть примерно так:

```
1  class Hash<K, V> {
2    LinkedList<V>[] items;
3    public void put(K key, V value) { ... }
4    public V get(K key) { ... }
5  }
```

`items` — массив связных списков, а `items[i]` — связный список всех объектов с ключами, которым сопоставляется индекс `i` (все коллизии для индекса `i`).

Вроде бы все работает... пока мы не задумаемся о коллизиях.

Допустим, у нас очень простая хеш-функция, которая использует длину строки:

```
1  int hashCodeOfKey(K key) {
2    return key.toString().length() % items.length;
3  }
```

Ключи `jim` и `bob` будут соответствовать одному индексу массива, хотя это разные ключи (с одинаковой длиной строки). Нам нужно произвести поиск по связному списку, чтобы найти правильные объекты, соответствующие ключам. Но как это сделать? Ведь в связном списке хранится значение, а не исходный ключ.

Одно из возможных решений — создание другого объекта (`Cell`), связывающего ключи со значениями. В этой реализации наш связный список будет иметь тип `Cell`.

Следующий код использует эту реализацию:

```
public class Hasher<K, V> {
  /* Класс узла связного списка. Используется только в хеш-таблице,
   * реализуется в виде двусвязного списка. */
  private static class LinkedListNode<K, V> {
    public LinkedListNode<K, V> next;
    public LinkedListNode<K, V> prev;
    public K key;
    public V value;
    public LinkedListNode(K k, V v) {
      key = k;
      value = v;
    }
  }

  private ArrayList<LinkedListNode<K, V>> arr;
  public Hasher(int capacity) {
    /* Создание списка связных списков. Список заполняется значениями
     * null (единственный способ создания массива заданного размера). */
    arr = new ArrayList<LinkedListNode<K, V>>();
    arr.ensureCapacity(capacity); // Optional optimization
    for (int i = 0; i < capacity; i++) {
      arr.add(null);
    }
  }

  /* Вставка ключа и значения в хеш-таблицу. */
  public void put(K key, V value) {
    LinkedListNode<K, V> node = getNodeForKey(key);
    if (node != null) { // Уже присутствует
      node.value = value; // Просто обновить значение.
      return;
    }

    node = new LinkedListNode<K, V>(key, value);
    int index = getIndexForKey(key);
    if (arr.get(index) != null) {
      node.next = arr.get(index);
      node.next.prev = node;
    }
    arr.set(index, node);
  }

  /* Удаление узла для ключа. */
  public void remove(K key) {
    LinkedListNode<K, V> node = getNodeForKey(key);
    if (node.prev != null) {
      node.prev.next = node.next;
```

```
48      } else {
49        /* Удаление начального узла - обновление. */
50        int hashKey = getIndexForKey(key);
51        arr.set(hashKey, node.next);
52      }
53
54      if (node.next != null) {
55        node.next.prev = node.prev;
56      }
57    }
58
59    /* Получение значения для ключа. */
60    public V get(K key) {
61      LinkedListNode<K, V> node = getNodeForKey(key);
62      return node == null ? null : node.value;
63    }
64
65    /* Получение узла связного списка для заданного ключа. */
66    private LinkedListNode<K, V> getNodeForKey(K key) {
67      int index = getIndexForKey(key);
68      LinkedListNode<K, V> current = arr.get(index);
69      while (current != null) {
70        if (current.key == key) {
71          return current;
72        }
73        current = current.next;
74      }
75      return null;
76    }
77
78    /* Очень наивная функция для связывания ключа с индексом. */
79    public int getIndexForKey(K key) {
80      return Math.abs(key.hashCode() % arr.size());
81    }
82  }
83
```

Аналогичная структура данных (отображение ключей на значения) может быть реализована на базе бинарного дерева поиска. Выборка элемента потребует не более $O(1)$ времени (хотя формально она займет более $O(1)$ при большом количестве коллизий), зато она избавляет от необходимости создавать излишне большой массив для хранения элементов.

8

Рекурсия и динамическое программирование

8.1. Ребенок поднимается по лестнице из n ступенек. За один шаг он может переместиться на одну, две или три ступеньки. Реализуйте метод, рассчитывающий количество возможных вариантов перемещения ребенка по лестнице.

РЕШЕНИЕ

Попробуем ответить на следующий вопрос: каким был самый последний прыжок — тот, который закончился на n-й ступеньке? Очевидно, это мог быть прыжок через 1, 2 или 3 ступеньки.

Сколькими способами можно добраться до n-й ступеньки? Этого мы пока не знаем, но задачу можно разложить на несколько подзадач.

Если задуматься, все пути к n-й ступеньке строятся из путей к трем предыдущим ступенькам. Добраться к n-й ступеньке можно любым из следующих способов:

- Добраться до (n-1)-й ступеньки и подняться на 1.
- Добраться до (n-2)-й ступеньки и подняться на 2.
- Добраться до (n-3)-й ступеньки и подняться на 3.

Нужно подсчитать количества таких путей и сложить их.

Будьте очень внимательны: многие разработчики почему-то пытаются умножать их. Умножение вариантов двух путей означает, что вы сначала выбираете один путь, а затем выбираете другой. Здесь происходит нечто иное.

Метод «грубой силы»

Существует довольно прямолинейный алгоритм с рекурсивной реализацией. Необходимо лишь следовать упомянутой выше логике вычисления количества путей:

```
countWays(n-1) + countWays(n-2) + countWays(n-3)
```

Некоторые трудности возникают с определением базового случая. Если еще не сделано ни одного шага, то сколько путей ведет к 0-й ступеньке — 0 или 1? Проще говоря, чему равно `countWays(0)` — 1 или 0?

Возможны оба варианта; единственно правильного ответа не существует. Тем не менее будет намного проще выбрать определение 1. Если выбрать определение 0, вам понадобятся дополнительные базовые случаи (или все кончится суммированием серий 0).

Простая реализация этого кода приведена ниже.

```
1  int countWays(int n) {
2    if (n < 0) {
3      return 0;
4    } else if (n == 0) {
5      return 1;
6    } else {
7      return countWays(n-1) + countWays(n-2) + countWays(n-3);
8    }
9  }
```

Как и в задаче о числах Фибоначчи, время выполнения этого алгоритма растет экспоненциально (приблизительно $O(3^N)$), поскольку каждый вызов разветвляется на три.

Решение с мемоизацией

В предыдущем решении `countWays` много раз вызывается для одних и тех же значений, а это неэффективно. Проблему можно решить применением мемоизации.

В двух словах, происходит следующее: если значение `n` уже встречалось ранее, то возвращается кэшированное значение. Каждый раз, когда вычисляется новое значение, оно добавляется в кэш.

Обычно для реализации кэша используется коллекция `HashMap<Integer, Integer>`. В нашем случае ключами будут целые числа от 1 до `n`, поэтому решение с целочисленным массивом получится более компактным.

```
1  int countWays(int n) {
2    int[] memo = new int[n + 1];
3    Arrays.fill(memo, -1);
4    return countWays(n, memo);
5  }
6
7  int countWays(int n, int[] memo) {
8    if (n < 0) {
9      return 0;
10   } else if (n == 0) {
11     return 1;
12   } else if (memo[n] > -1) {
13     return memo[n];
14   } else {
15     memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
16               countWays(n - 3, memo);
17     return memo[n];
18   }
19 }
```

Независимо от того, применяется мемоизация или нет, количество путей быстро выходит за границы целочисленного типа. Всего лишь для `n=37` уже возникает переполнение. Использование типа `long` отложит проблему, но не решит ее полностью.

Будет замечательно, если вы упомянете об этой проблеме интервьюеру. Вряд ли он заставит вас искать обходное решение (хотя и это можно сделать с помощью класса `BigInteger`), но вам стоит показать, что вы учитываете подобные аспекты.

8.2. Робот стоит в левом верхнем углу сетки, состоящей из r строк и c столбцов. Робот может перемещаться в двух направлениях: вправо и вниз, но некоторые ячейки сетки заблокированы, то есть робот через них проходить не может. Разработайте алгоритм построения маршрута от левого верхнего до правого нижнего угла.

РЕШЕНИЕ

Если представить себе сетку, переход к ячейке (r,c) возможен только после перехода к одной из смежных ячеек: (r-1,c) или (r,c-1). Итак, нам нужно найти путь к (r-1,c) или (r,c-1).

Как найти путь к одной из этих ячеек? Чтобы перейти к ячейке (r-1,c) или (r,c-1), необходимо перейти к одной из смежных ячеек. Следовательно, нужно найти путь к ячейке, смежной с (r-1,c), то есть к ячейке с координатами (r-2,c) и (r-1,c-1), или же к ячейке, смежной с (r,c-1), то есть к ячейке с координатами (r-1,c-1) и (r,c-2).

Обратите внимание: ячейка (r-1,c-1) упоминается дважды; мы обсудим эту проблему позднее.

При работе с двумерными массивами часто используются имена переменных x и y. На самом деле это может привести к ошибкам. Люди склонны рассматривать x как первую, а y — как вторую координату в матрице (matrix[x][y]). Но на самом деле это неверно. Первая координата обычно рассматривается как номер строки, которой в действительности соответствует значение y (строки нумеруются по вертикали!). Используйте запись matrix[y][x]. Или просто не создавайте себе проблем и используйте имена переменных r (для строк) и c (для столбцов).

Итак, чтобы найти путь от исходного квадрата, мы будем двигаться в обратном направлении: начиная с последней ячейки, будем искать путь к каждому смежной ячейке. Далее приведена рекурсивная реализация нашего алгоритма.

```
1  ArrayList<Point> getPath(boolean[][] maze) {
2    if (maze == null || maze.length == 0) return null;
3    ArrayList<Point> path = new ArrayList<Point>();
4    if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
5      return path;
6    }
7    return null;
8  }
9
10 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
11   /* При выходе за границы или недоступности вернуть управление.*/
12   if (col < 0 || row < 0 || !maze[row][col]) {
13     return false;
14   }
15
16   boolean isAtOrigin = (row == 0) && (col == 0);
17
18   /* Если существует путь от начала до текущей ячейки, добавить. */
19   if (isAtOrigin || getPath(maze, row, col - 1, path) ||
20       getPath(maze, row - 1, col, path)) {
21     Point p = new Point(row, col);
22     path.add(p);
23     return true;
24   }
25
26   return false;
27 }
```

Решение имеет сложность $O(2^{r+c})$, потому что каждый путь состоит из r+c шагов и на каждом шаге выбирается один из двух вариантов.

Нельзя ли найти более быстрый способ? Часто экспоненциальные алгоритмы удается оптимизировать за счет исключения повторной работы. Какая работа здесь повторяется?

Анализ алгоритма показывает, что некоторые ячейки посещаются многократно. Собственно, многократно посещается каждая ячейка — ведь сетка состоит из `rc` ячеек, но мы проделываем работу $O(2^{r+c})$. Если бы каждая ячейка посещалась всего один раз, вероятно, алгоритм имел бы временную сложность $O(rc)$ (если только каждое посещение не требовало бы аномального объема работы).

Как работает текущий алгоритм? Чтобы найти путь к `(r,c)`, мы ищем путь к ячейке со смежной координатой: `(r-1,c)` или `(r,c-1)`. Конечно, если одна из таких ячеек выходит за границы сетки, она игнорируется. Затем проверяются смежные координаты: `(r-2,c)`, `(r-1,c-1)`, `(r-1,c-1)` и `(r,c-2)`. Ячейка `(r-1,c-1)` встречается дважды, а это свидетельствует о дублировании работы. В идеале следовало бы запомнить, что ячейка `(r-1,c-1)` уже посещалась, чтобы не тратить на нее время.

На этом принципе построен алгоритм динамического программирования, приведенный ниже.

```
1  ArrayList<Point> getPath(boolean[][] maze) {
2    if (maze == null || maze.length == 0) return null;
3
4    ArrayList<Point> path = new ArrayList<Point>();
5    HashMap<Point, Boolean> cache = new HashMap<Point, Boolean>();
6    int lastRow = maze.length - 1;
7    int lastCol = maze[0].length - 1;
8    if (getPath(maze, lastRow, lastCol, path, cache)) {
9      return path;
10   }
11   return null;
12 }
13
14 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
15                  HashMap<Point, Boolean> cache) {
16   /* При выходе за границы или недоступности вернуть управление.*/
17   if (col < 0 || row < 0 || !maze[row][col]) {
18     return false;
19   }
20   Point p = new Point(row, col);
21
22   /* Если ячейка уже посещалась, вернуть управление. */
23   if (cache.containsKey(p)) {
24     return cache.get(p);
25   }
26
27   boolean isAtOrigin = (row == 0) && (col == 0);
28   boolean success = false;
29
30   /* Если существует путь от начала к текущей позиции, добавить
31    * текущую позицию.*/
32   if (isAtOrigin || getPath(maze, row, col - 1, path, cache) ||
33                     getPath(maze, row - 1, col, path, cache)) {
34     path.add(p);
35     success = true;
36   }
```

```
37
38    cache.put(p, success); // Кэширование результата
39    return success;
40  }
```

Это простое изменение существенно ускорит выполнение кода. Алгоритм теперь выполняется за время $O(XY)$, потому что каждая ячейка посещается всего один раз.

8.3. Определим «волшебный» индекс для массива `A[0...n-1]` как индекс, для которого выполняется условие `A[i]=i`. Для заданного отсортированного массива, не содержащего одинаковых значений, напишите метод поиска «волшебного» индекса в массиве `A` (если он существует).

Дополнительно

Что произойдет, если массив может содержать одинаковые значения?

РЕШЕНИЕ

Первым в голову приходит решение методом «грубой силы» — нет ничего постыдного в том, чтобы упомянуть о нем. Достаточно перебрать элементы массива и найти элемент, соответствующий условию.

```
1  int magicSlow(int[] array) {
2    for (int i = 0; i < array.length; i++) {
3      if (array[i] == i) {
4        return i;
5     }
6    }
7    return -1;
8  }
```

Впрочем, в формулировке задачи указано, что массив отсортирован, — скорее всего, мы должны как-то использовать этот факт.

Нетрудно понять, что эта задача имеет много общего с классической задачей бинарного поиска. Как же применить методы бинарного поиска в данном случае?

При бинарном поиске элемента `k` алгоритм сравнивает его значение со средним элементом `x`, после чего определяет, находится `k` слева или справа от `x`.

Если взять этот метод за основу, можно ли определить местонахождение «волшебного» интекса на основании значения среднего элемента? Рассмотрим пример массива:

-40	-20	-1	1	2	**3**	5	7	9	12	13
0	1	2	3	4	**5**	6	7	8	9	10

Проверяя средний элемент `A[5] = 3`, мы знаем, что «волшебный» индекс должен находиться справа, так как `A[mid] < mid`.

Почему «волшебный» индекс не может находиться в левой части? При переходе от `i` к `i` – 1 значение индекса должно уменьшиться на 1, если не более (так как массив отсортирован, а все его элементы различны). Итак, если средний элемент уже слишком мал, чтобы быть «волшебным» индексом, при перемещении влево со смещением на `k` индексов и (как минимум) на `k` значений все последующие элементы тоже будут слишком малы.

Код, полученный в результате применения этого рекурсивного алгоритма, очень похож на бинарный поиск.

```
1  int magicFast(int[] array) {
2    return magicFast(array, 0, array.length - 1);
3  }
4
5  int magicFast(int[] array, int start, int end) {
6    if (end < start) {
7      return -1;
8    }
9    int mid = (start + end) / 2;
10   if (array[mid] == mid) {
11     return mid;
12   } else if (array[mid] > mid){
13     return magicFast(array, start, mid - 1);
14   } else {
15     return magicFast(array, mid + 1, end);
16   }
17 }
```

Дополнительно: а если элементы повторяются?

Если в массиве имеются одинаковые элементы, то алгоритм работать не будет. Рассмотрим следующий массив:

-10	-5	2	2	2	**3**	4	7	9	12	13
0	1	2	3	4	**5**	6	7	8	9	10

Если `A[mid] < mid`, мы не можем решить, где находится «волшебный» элемент. Он может быть расположен как справа, так и слева (где он фактически и находится).

Может ли он находиться *где угодно* слева? Нет. Так как `A[5] = 3`, мы знаем, что `A[4]` не может быть «волшебным» элементом. Чтобы элемент `A[4]` был «волшебным» индексом, он должен быть равен 4, но в то же время мы знаем, что `A[4]` должен быть меньше либо равен `A[5]`.

Фактически, когда мы видим, что `A[5] = 3`, следует провести рекурсивный поиск в правой части, как это и делалось раньше. Но чтобы найти элемент в левой части, можно пропустить группу элементов и произвести поиск только среди `A[0] - A[3]`, где `A[3]` — первый элемент, который может быть «волшебным».

В общем, нам нужно сравнить `midIndex` и `midValue`. Если они не совпадают, то поиск проводится в левой и правой части:

- левая сторона: поиск среди элементов от `start` до `Math.min(midIndex - 1, midValue)`;
- правая сторона: поиск среди элементов от `Math.Max(midIndex + 1, midValue)` до `end`.

Ниже приведена реализация этого алгоритма:

```
1  int magicFast(int[] array) {
2    return magicFast(array, 0, array.length - 1);
3  }
4
5  int magicFast(int[] array, int start, int end) {
6    if (end < start) return -1;
```

```
7
8    int midIndex = (start + end) / 2;
9    int midValue = array[midIndex];
10    if (midValue == midIndex) {
11      return midIndex;
12    }
13
14    /* Поиск в левой части */
15    int leftIndex = Math.min(midIndex - 1, midValue);
16    int left = magicFast(array, start, leftIndex);
17    if (left >= 0) {
18      return left;
19    }
20
21    /* Поиск в правой части */
22    int rightIndex = Math.max(midIndex + 1, midValue);
23    int right = magicFast(array, rightIndex, end);
24
25    return right;
26  }
```

Обратите внимание: если в этом коде все элементы различны, то метод практически идентичен первому решению.

8.4. Напишите метод, возвращающий все подмножества заданного множества.

РЕШЕНИЕ

Сначала определимся с разумными оценками временной и пространственной сложности.

Сколько подмножеств существует у заданного множества? При генерировании подмножества каждый элемент может либо находиться в подмножестве, либо нет. Пройдясь по всем элементам, мы получаем {2 * 2 * ...} n раз, или 2^n подмножества.

Предполагая, что программа будет возвращать список подмножеств, время для лучшего случая равно общему количеству элементов во всех подмножествах. Существуют 2^n подмножества, каждый из n элементов присутствует в половине подмножеств (2^{n-1}). Таким образом, общее количество элементов во всех подмножествах равно $n*2^{n-1}$.

Итак, нам не добиться лучшей сложности по времени/затратам памяти, чем $O(n\,2^n)$.

Совокупность множеств $\{a_1, a_2, \ldots, a_n\}$ принято называть *степенным множеством* и обозначать $P(\{a_1, a_2, \ldots, a_n\})$, или просто $P(n)$.

Решение 1. Рекурсивное

Эта задача идеально подходит для метода «Базовый случай с расширением». Представьте, что мы пытаемся найти все подмножества множества $S = \{a_1, a_2, \ldots, a_n\}$. Начнем с базового случая.

$n = 0$

Одно подмножество пустого множества: {}

$n = 1$

Множество $\{a_1\}$ имеет два подмножества: $\{\}$, $\{a_1\}$

$n = 2$

Множество $\{a_1, a_2\}$ имеет четыре подмножества: $\{\}$, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$

$n = 3$

Теперь самое интересное. Нам нужно найти способ генерации решения для `n = 3`, основанный на предыдущих решениях.

Чем отличаются решения для `n = 3` и `n = 2`? Рассмотрим их более подробно:

```
P(2) = {}, {a1}, {a2}, {a1, a2}
P(3) = {}, {a1}, {a2}, {a3}, {a1, a2}, {a1, a3}, {a2, a3}, {a1, a2, a3}
```

Разница между этими решениями в том, что в `P(2)` нет подмножеств, содержащих a_3:

```
P(3) - P(2) = {a3}, {a1, a3}, {a2, a3}, {a1, a2, a3}
```

Как использовать `P(2)` для создания `P(3)`? Можно клонировать подмножества из `P(2)` и добавить к ним a_3:

```
P(2) = {} , {a1}, {a2}, {a1, a2}
P(2) + a3  = {a3}, {a1, a3}, {a2, a3}, {a1, a2, a3}
```

Объединив эти две строки, получаем `P(3)`.

$n > 0$

Построение решения `P(n)` для общего случая — простое обобщение приведенных выше шагов. Мы вычисляем `P(n-1)`, клонируем результаты и добавляем a_n в каждое из клонированных множеств.

Ниже приведена реализация этого алгоритма.

```
1  ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
2    ArrayList<ArrayList<Integer>> allsubsets;
3    if (set.size() == index) { // Базовый случай - добавить пустое множество
4      allsubsets = new ArrayList<ArrayList<Integer>>();
5      allsubsets.add(new ArrayList<Integer>()); // Пустое множество
6    } else {
7      allsubsets = getSubsets(set, index + 1);
8      int item = set.get(index);
9      ArrayList<ArrayList<Integer>> moresubsets =
10        new ArrayList<ArrayList<Integer>>();
11     for (ArrayList<Integer> subset : allsubsets) {
12       ArrayList<Integer> newsubset = new ArrayList<Integer>();
13       newsubset.addAll(subset); //
14       newsubset.add(item);
15       moresubsets.add(newsubset);
16     }
17      allsubsets.addAll(moresubsets);
18   }
19   return allsubsets;
20 }
```

Это решение потребует затрат времени/пространства $O(2n)$ и будет наилучшим из всех возможных. Чтобы добиться небольшой оптимизации, можно также реализовать итерационную версию алгоритма.

Решение 2. Комбинаторика

Предыдущее решение работает нормально, но мы рассмотрим еще один вариант.

Вспомните, что при генерировании множества каждый элемент может: (1) принадлежать подмножеству (состояние «да») и (2) не принадлежать подмножеству (состояние «нет»). Это означает, что каждое подмножество является последовательностью значений «да»/«нет», например «да, да, нет, нет, да, нет».

Таким образом, мы получаем $2n$ возможных подмножеств. Как пройтись по всем возможным последовательностям состояний «да»/«нет» всех элементов? Если каждое «да» рассматривать как 1, а каждое «нет» — как 0, то любое подмножество можно представить в виде двоичной строки.

Таким образом, построение всех подмножеств сводится к получению всех двоичных (целых) чисел. Мы перебираем все числа от 0 до $2n$ и переводим двоичное представление числа в множество. Элементарно!

```
1  ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
2    ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integ
er>>();
3    int max = 1 << set.size(); /* Вычисление 2^n */
4    for (int k = 0; k < max; k++) {
5      ArrayList<Integer> subset = convertIntToSet(k, set);
6      allsubsets.add(subset);
7    }
8    return allsubsets;
9  }
10
11  ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
12    ArrayList<Integer> subset = new ArrayList<Integer>();
13    int index = 0;
14    for (int k = x; k > 0; k >>= 1) {
15      if ((k & 1) == 1) {
16        subset.add(set.get(index));
17      }
18      index++;
19    }
20    return subset;
21  }
```

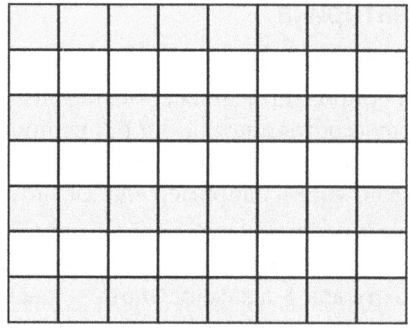
Это решение ничем не лучше и не хуже предыдущего.

8.5. Напишите рекурсивную функцию для умножения двух положительных целых чисел без использования оператора *. Допускается использование операций сложения, вычитания и поразрядного сдвига, но их количество должно быть минимальным.

РЕШЕНИЕ

Давайте задумаемся, что же это означает — выполнить умножение?

Этот прием эффективно работает на многих вопросах собеседования. Часто бывает полезно задуматься над тем, что же на самом деле происходит при выполнении некоторой операции, даже если на первый взгляд это очевидно.



Умножение 8×7 можно представить в виде 8+8+8+8+8+8+8 (или суммировании 7 восемь раз). Также можно представить результат как количество ячеек в таблице 8×7.

Решение 1

Как подсчитать количество ячеек в таблице? Можно подсчитать каждую ячейку по отдельности, но это будет слишком медленно.

Также можно подсчитать половину ячеек и удвоить результат (то есть сложить величину с ней самой). Чтобы подсчитать половину ячеек, можно повторить этот же процесс.

Конечно, прием с «удвоением» работает только для четных чисел. Если число нечетно, подсчет/суммирование приходится выполнять с нуля.

```
1  int minProduct(int a, int b) {
2    int bigger = a < b ? b : a;
3    int smaller = a < b ? a : b;
4    return minProductHelper(smaller, bigger);
5  }
6
7  int minProductHelper(int smaller, int bigger) {
8    if (smaller == 0) { // 0 x bigger = 0
9      return 0;
10   } else if (smaller == 1) { // 1 x bigger = bigger
11     return bigger;
12   }
13   /* Вычисление половины. Если значение нечетно, вычислить другую
14    * половину, если четно - удвоить. */
15   int s = smaller >> 1; // Разделить на 2
16   int side1 = minProduct(s, bigger);
17   int side2 = side1;
18   if (smaller % 2 == 1) {
19     side2 = minProductHelper(smaller - s, bigger);
20   }
21
22   return side1 + side2;
23 }
```

Можно ли действовать более эффективно? Да.

Решение 2

Если понаблюдать за тем, как происходит рекурсия, можно заметить, что часть работы повторяется. Пример:

```
minProduct(17, 23)
   minProduct(8, 23)
      minProduct(4, 23) * 2
      ...
   + minProduct(9, 23)
      minProduct(4, 23)
      ...
   + minProduct(5, 23)
      ...
```

Второй вызов `minProduct(4, 23)` ничего не знает о предыдущем, поэтому он повторяет ту же работу. Полученный результат следовало бы кэшировать.

```
1  int minProduct(int a, int b) {
2    int bigger = a < b ? b : a;
3    int smaller = a < b ? a : b;
4
5    int memo[] = new int[smaller + 1];
6    return minProduct(smaller, bigger, memo);
7  }
8
9  int minProduct(int smaller, int bigger, int[] memo) {
10   if (smaller == 0) {
11     return 0;
12   } else if (smaller == 1) {
13     return bigger;
14   } else if (memo[smaller] > 0) {
15     return memo[smaller];
16   }
17   /* Вычисление половины. Если значение нечетно, вычислить другую
18    * половину, если четно - удвоить. */
19   int s = smaller >> 1; // Разделить на 2
20   int side1 = minProduct(s, bigger, memo); // Вычислить половину
21   int side2 = side1;
22   if (smaller % 2 == 1) {
23     side2 = minProduct(smaller - s, bigger, memo);
24   }
25
26   /* Суммирование и кэширование.*/
27   memo[smaller] = side1 + side2;
28   return memo[smaller];
29 }
```

Впрочем, и эту реализацию можно немного ускорить.

Решение 3

Присмотревшись к этому коду, можно заметить, что вызов `minProduct` для четных чисел работает намного быстрее, чем для нечетных. Например, если вызвать `minProduct(30, 35)`, мы просто вызываем `minProduct(15, 35)` и удваиваем результат.

С другой стороны, при вызове `minProduct(31, 35)` приходится вызывать `minProduct(15, 35)` и `minProduct(16, 35)`.

И это лишнее. Можно действовать иначе:

```
minProduct(31, 35) = 2 * minProduct(15, 35) + 35
```

В конце концов, если 31 = 2*15+1, то 31×35 = 2*15*35+35.

Логика окончательного решения строится на том, что для нечетных чисел мы просто делим `smaller` на 2 и удваиваем результат рекурсивного вызова. Для нечетных чисел делается то же самое, но к результату добавляется `bigger`.

Это изменение имеет неожиданное положительное следствие. Функция `minProduct` просто осуществляет нисходящую рекурсию с постоянным уменьшением чисел. Вызовы никогда не повторяются, поэтому надобность в кэшировании отпадает.

```
1  int minProduct(int a, int b) {
2    int bigger = a < b ? b : a;
3    int smaller = a < b ? a : b;
4    return minProductHelper(smaller, bigger);
5  }
6
7  int minProductHelper(int smaller, int bigger) {
8    if (smaller == 0) return 0;
9    else if (smaller == 1) return bigger;
10
11    int s = smaller >> 1; // Разделить на 2
12    int halfProd = minProductHelper(s, bigger);
13
14    if (smaller % 2 == 0) {
15      return halfProd + halfProd;
16    } else {
17      return halfProd + halfProd + bigger;
18    }
19  }
```

Алгоритм выполняется за время $O(\log S)$, где S — меньшее из двух чисел.

8.6. В классической задаче о ханойских башнях имеются 3 башни и N дисков разного размера, которые могут перекладываться между башнями. В начале головоломки диски отсортированы по возрастанию размера сверху вниз (то есть каждый диск лежит на диске большего размера). Установлены следующие ограничения:

(1) За один раз можно переместить только один диск.

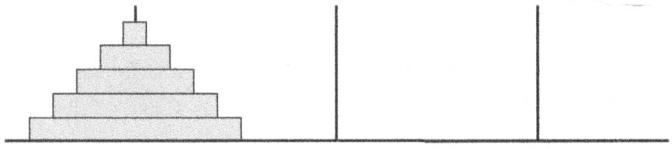
(2) Диск кладется на вершину другой башни.

(3) Диск нельзя положить на диск меньшего размера.

Напишите программу для перемещения дисков с первой башни на последнюю с использованием стеков.

РЕШЕНИЕ

Данная задача хорошо подходит для метода «Базовый случай с расширением».



Начнем с минимально возможного случая `n = 1`.

***n* = 1**. Возможно ли просто переложить Диск 1 с Башни 1 на Башню 3? Да.

1. Диск 1 перекладывается с Башни 1 на Башню 3.

***n* = 2**. Возможно ли переместить Диски 1 и 2 с Башни 1 на Башню 3? Да.

2. Переместить Диск 1 с Башни 1 на Башню 2.
3. Переместить Диск 2 с Башни 1 на Башню 3.
4. Переместить Диск 1 с Башни 2 на Башню 3.

Башня 2 служит буфером, через который мы перемещаем другие диски на Башню 3.

***n* = 3**. Возможно ли переместить Диски 1, 2 и 3 с Башни 1 на Башню 3? Да.

1. Мы знаем, что можем переместить два верхних диска с одной башни на другую (как описано выше). Будем считать, что это уже сделано, но в этом случае они перемещаются на Башню 2.
2. Переместить Диск 3 на Башню 3.
3. Переместить Диск 1 и Диск 2 на Башню 3. Мы уже знаем, как это сделать, — просто повторяем то, что мы сделали на шаге 1.

Случай *n* = 4. Возможно ли переместить Диски 1, 2, 3 и 4 с Башни 1 на Башню 3? Да.

1. Перемещаем Диски 1, 2 и 3 на Башню 2. Вы уже знаете, как это делается, из предыдущих примеров.
2. Перемещаем Диск 4 на Башню 3.
3. Перемещаем Диски 1, 2 и 3 обратно на Башню 3.

Помните, что обозначения «Башня 2» и «Башня 3» несущественны. Эти башни эквивалентны. Перемещение дисков на Башню 3 (с использованием Башни 2 в качестве буфера) эквивалентно перемещению дисков на Башню 2 с использованием Башни 3 в качестве буфера.

Данный подход сводится к рекурсивному алгоритму. Каждый раз выполняются действия, описанные следующим псевдокодом:

```
1  moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
2    /* Базовый случай */
3    if (n <= 0) return;
4
5    /* Переместить верхние n-1 дисков с origin на buffer, используя destination
           как буфер. */
6    moveDisks(n - 1, origin, buffer, destination);
7
8    /* Переместить верхний диск с origin на destination
9    moveTop(origin, destination);
```

```
10
11    /* Переместить верхние n-1 дисков с buffer на destination, используя origin
             как буфер. */
12    moveDisks(n - 1, buffer, destination, origin);
13  }
```

Далее приводится более подробная реализация того же алгоритма с использованием объектно-ориентированного подхода:

```
1   void main(String[] args) {
2     int n = 3;
3     Tower[] towers = new Tower[n];
4     for (int i = 0; i < 3; i++) {
5       towers[i] = new Tower(i);
6     }
7
8     for (int i = n - 1; i >= 0; i--) {
9       towers[0].add(i);
10    }
11    towers[0].moveDisks(n, towers[2], towers[1]);
12  }
13
14  class Tower {
15    private Stack<Integer> disks;
16    private int index;
17    public Tower(int i) {
18      disks = new Stack<Integer>();
19      index = i;
20    }
21
22    public int index() {
23      return index;
24    }
25
26    public void add(int d) {
27      if (!disks.isEmpty() && disks.peek() <= d) {
28        System.out.println("Error placing disk " + d);
29      } else {
30        disks.push(d);
31     }
32    }
33
34    public void moveTopTo(Tower t) {
35      int top = disks.pop();
36      t.add(top);
37    }
38
39     public void moveDisks(int n, Tower destination, Tower buffer) {
40      if (n > 0) {
41        moveDisks(n - 1, buffer, destination);
42        moveTopTo(destination);
43        buffer.moveDisks(n - 1, destination, this);
44     }
45    }
46  }
```

Реализация башен как самостоятельных объектов не является строго необходимой, но делает код проще и понятнее.

8.7. Напишите метод для вычисления всех перестановок строки, состоящей из уникальных символов.

РЕШЕНИЕ

Как и во многих других рекурсивных задачах, здесь хорошо сработает метод «Базовый случай с расширением». Допустим, имеется строка S, состоящая из символов $a_1a_2...a_n$.

Решение 1. Перестановки первых *n*–1 символов

Базовый случай: перестановки подстроки, состоящей из первого символа

Единственная перестановка a_1 — это строка a_1. Имеем:

$P(a_1) = a_1$

Случай: перестановки a_1a_2

$P(a_1a_2) = a_1a_2$ и a_2a_1

Случай: перестановки $a_1a_2a_3$

$P(a_1a_2a_3) = a_1a_2a_3,\ a_1a_3a_2,\ a_2a_1a_3,\ a_2a_3a_1,\ a_3a_1a_2,\ a_3a_2a_1$

Случай: перестановки $a_1a_2a_3a_4$

Первый нетривиальный случай. Как сгенерировать все перестановки $a_1a_2a_3a_4$, если известны перестановки $a_1a_2a_3$?

Каждая перестановка $a_1a_2a_3a_4$ представляет вариант упорядочения $a_1a_2a_3$. Например, $a_2a_4a_1a_3$ представляет порядок $a_2a_1a_3$. Следовательно, если взять перестановки $a_1a_2a_3$ и добавить a_4 во все возможные позиции, мы получим все перестановки $a_1a_2a_3a_4$.

Этот алгоритм можно реализовать рекурсивно:

```
1  ArrayList<String> getPerms(String str) {
2    if (str == null) return null;
3
4    ArrayList<String> permutations = new ArrayList<String>();
5    if (str.length() == 0) { // Базовый случай
6      permutations.add("");
7      return permutations;
8    }
9
10    char first = str.charAt(0); // Получить первый символ
11    String remainder = str.substring(1); // Удалить первый символ
12    ArrayList<String> words = getPerms(remainder);
13    for (String word : words) {
14      for (int j = 0; j <= word.length(); j++) {
15        String s = insertCharAt(word, first, j);
16        permutations.add(s);
17     }
18    }
19    return permutations;
20  }
21
```

```
22  /* Вставить символ с индексом i в слово. */
23  String insertCharAt(String word, char c, int i) {
24    String start = word.substring(0, i);
25    String end = word.substring(i);
26    return start + c + end;
27  }
```

Алгоритм выполняется за время $O(n!)$.

Решение 2. Перестановки всех строк из n–1 символов

Базовый случай: строка из одного символа

Единственная перестановка a_1 — это строка a_1. Имеем:

$P(a_1) = a_1$

Случай: строки из двух символов

$P(a_1a_2) = a_1a_2$ и a_2a_1

$P(a_2a_3) = a_2a_3$ и a_3a_2

$P(a_1a_3) = a_1a_3$ и a_3a_1

Случай: строки из трех символов

Этот случай уже интереснее. Как сгенерировать все перестановки строк из трех символов (например, $a_1a_2a_3$), если известны перестановки строк из двух символов?

По сути, нужно просто «опробовать» каждый символ в качестве первого, а затем объединить перестановки.

$P(a_1a_2a_3) = \{a_1 + P(a_2a_3)\} + \{a_2 + P(a_1a_3)\} + \{a_3 + P(a_1a_2)\}$

$\{a_1 + P(a_2a_3)\} \rightarrow a_1a_2a_3, a_1a_3a_2$

$\{a_2 + P(a_1a_3)\} \rightarrow a_2a_1a_3, a_2a_3a_1$

$\{a_3 + P(a_1a_2)\} \rightarrow a_3a_1a_2, a_3a_2a_1$

Аналогичным образом на базе всех перестановок строк из трех символов генерируются перестановки строк из четырех символов, и т. д. Алгоритм реализуется достаточно прямолинейно:

```
1  ArrayList<String> getPerms(String remainder) {
2    int len = remainder.length();
3    ArrayList<String> result = new ArrayList<String>();
4
5    /* Базовый случай. */
6    if (len == 0) {
7      result.add(""); // Обязательно верните пустую строку!
8      return result;
9    }
10
11
12    for (int i = 0; i < len; i++) {
13      /* Удалить символ i и найти перестановки оставшихся символов.*/
14      String before = remainder.substring(0, i);
15      String after = remainder.substring(i + 1, len);
16      ArrayList<String> partials = getPerms(before + after);
17
18      /* Присоединить символ i перед каждой перестановкой.*/
```

```
19      for (String s : partials) {
20        result.add(remainder.charAt(i) + s);
21      }
22    }
23
24    return result;
25  }
```

Также вместо того, чтобы передавать перестановки вверх по стеку, можно продвинуть префикс вниз по стеку. При достижении низа (базовый случай) `prefix` содержит полную перестановку.

```
1  ArrayList<String> getPerms(String str) {
2    ArrayList<String> result = new ArrayList<String>();
3    getPerms("", str, result);
4    return result;
5  }
6
7  void getPerms(String prefix, String remainder, ArrayList<String> result) {
8    if (remainder.length() == 0) result.add(prefix);
9
10   int len = remainder.length();
11   for (int i = 0; i < len; i++) {
12     String before = remainder.substring(0, i);
13     String after = remainder.substring(i + 1, len);
14     char c = remainder.charAt(i);
15     getPerms(prefix + c, before + after, result);
16   }
17 }
```

Оба решения выполняются за время $O(n!)$. Так как всего существует $n!$ перестановок, улучшить этот результат не удастся.

8.8. Напишите метод для вычисления всех перестановок строки, символы которой не обязаны быть уникальными. В списке перестановок дубликатов быть не должно.

РЕШЕНИЕ

Эта задача очень похожа на предыдущую, если не считать того, что в слове могут присутствовать повторяющиеся символы.

Одно из простых решений — проделать ту же работу, проверить, создавалась ли полученная перестановка ранее, и если не создавалась, — добавить ее в список. Для этого достаточно простой хеш-таблицы. Это решение будет выполняться за время $O(n!)$ в худшем случае (а на самом деле во всех случаях).

И хотя превзойти это худшее время не удастся, алгоритм можно спроектировать так, чтобы он превосходил его во многих случаях. Рассмотрим строку, состоящую из одних дубликатов (например, *aaaaaaaaaaaaaa*). Ее обработка займет очень много времени, так как у строки из 13 символов существует более 6 миллиардов перестановок, хотя уникальная перестановка всего одна.

В идеале нам хотелось бы генерировать только уникальные перестановки вместо того, чтобы создавать все возможные перестановки, а затем убирать дубликаты.

Исходными данными станет количество вхождений каждой буквы (это делается достаточно просто — используйте хеш-таблицу). Для такой строки, как *aabbbbc*, данные будут выглядеть так:

```
a->2 | b->4 | c->1
```

Теперь представьте, что вам потребовалось сгенерировать перестановку этой строки (представленной в виде хеш-таблицы). Прежде всего нужно решить, какой символ станет первым — *a*, *b* или *c*. После этого необходимо решить новую подзадачу: найти все перестановки оставшихся символов и присоединить их к уже выбранному «префиксу».

```
P(a->2 | b->4 | c->1) = {a + P(a->1 | b->4 | c->1)} +
                        {b + P(a->2 | b->3 | c->1)} +
                        {c + P(a->2 | b->4 | c->0)}
  P(a->1 | b->4 | c->1) = {a + P(a->0 | b->4 | c->1)} +
                           {b + P(a->1 | b->3 | c->1)} +
                           {c + P(a->1 | b->4 | c->0)}
  P(a->2 | b->3 | c->1) = {a + P(a->1 | b->3 | c->1)} +
                           {b + P(a->2 | b->2 | c->1)} +
                           {c + P(a->2 | b->3 | c->0)}
  P(a->2 | b->4 | c->0) = {a + P(a->1 | b->4 | c->0)} +
                           {b + P(a->2 | b->3 | c->0)}
```

Со временем мы придем к ситуации, в которой не осталось ни одного символа.

Ниже приведена реализация этого алгоритма.

```
1  ArrayList<String> printPerms(String s) {
2    ArrayList<String> result = new ArrayList<String>();
3    HashMap<Character, Integer> map = buildFreqTable(s);
4    printPerms(map, "", s.length(), result);
5    return result;
6  }
7
8  HashMap<Character, Integer> buildFreqTable(String s) {
9    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
10    for (char c : s.toCharArray()) {
11      if (!map.containsKey(c)) {
12        map.put(c, 0);
13     }
14      map.put(c, map.get(c) + 1);
15    }
16    return map;
17  }
18
19  void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,
20                  ArrayList<String> result) {
21    /* Базовый случай. Перестановка завершена. */
22    if (remaining == 0) {
23      result.add(prefix);
24      return;
25    }
26
27    /* Генерирование остальных перестановок. */
28    for (Character c : map.keySet()) {
```

```
29      int count = map.get(c);
30      if (count > 0) {
31        map.put(c,  count - 1);
32        printPerms(map, prefix + c, remaining - 1, result);
33        map.put(c,  count);
34      }
35    }
36  }
```

Для строк, содержащих большое количество дубликатов, этот алгоритм будет выполняться намного быстрее предыдущей версии.

8.9. Реализуйте алгоритм для вывода всех корректных (правильно открытых и закрытых) комбинаций из n пар круглых скобок.

Пример:

Ввод: 3

Вывод: ((())), (()()), (())(), ()(()), ()()()

РЕШЕНИЕ

Первое, что приходит в голову, — использовать рекурсивный подход и строить решение для `f(n)`, добавляя пары круглых скобок в `f(n-1)`. Мысль абсолютно здравая.

Рассмотрим решение для `n = 3`:

```
(()())    ((()))    ()(())    (())()    ()()()
```

Как получить это решение из решения для `n = 2`?

```
(())      ()()
```

Можно вставить пары скобок в каждую существующую пару скобок, а также одну пару в начале строки. Другие возможные места для вставки скобок (например, в конце строки) сводятся к предыдущим случаям.

Итак, имеем следующее:

```
(()) -> (()()) /* скобки вставлены после первой левой скобки */
     -> ((())) /* скобки вставлены после второй левой скобки */
     -> ()(()) /* скобки вставлены в начале строки */
()() -> (())() /* скобки вставлены после первой левой скобки */
     -> ()(()) /* скобки вставлены после второй левой скобки */
     -> ()()() /* скобки вставлены в начале строки */
```

Постойте, в списке есть дубликаты! Строка `()(())` встречается дважды!

Если мы будем использовать данный подход, следует реализовать проверку дубликатов перед добавлением строки в список.

```
1  Set<String> generateParens(int remaining) {
2    Set<String> set = new HashSet<String>();
3    if (remaining == 0) {
4      set.add("");
5    } else {
6      Set<String> prev = generateParens(remaining - 1);
7      for (String str : prev) {
8        for (int i = 0; i < str.length(); i++) {
9          if (str.charAt(i) == '(') {
```

```
            String s = insertInside(str, i);
            /* Добавить строку s в множество, если ее там нет. Примечание:
             * HashSet автоматически проверяет дубликаты перед
             * добавлением, так что явная проверка окажется лишней. */
            set.add(s);
        }
       }
         set.add("()" + str);
     }
    }
    return set;
}

String insertInside(String str, int leftIndex) {
  String left = str.substring(0, leftIndex + 1);
  String right = str.substring(leftIndex + 1, str.length());
  return left + "()" + right;
}
```

Алгоритм работает, но не очень эффективно. Он тратит слишком много времени на дублирующиеся строки.

Проблемы дублирования можно избежать путем построения строки с нуля. В этом случае левые и правые скобки добавляются до тех пор, пока выражение остается корректным.

При каждом рекурсивном вызове мы получаем индекс определенного символа в строке. Теперь нужно выбрать скобку (левую или правую). Когда можно использовать левую скобку, а когда — правую?

1. *Левая скобка*: пока не израсходованы все левые скобки, мы всегда можем вставить левую скобку.
2. *Правая скобка*: добавить правую скобку можно в том случае, если добавление не приведет к синтаксической ошибке. Когда появляется синтаксическая ошибка? Тогда, когда правых скобок больше, чем левых.

Таким образом, нам нужно отслеживать количество открывающих и закрывающих скобок. Если в строку можно вставить левую скобку, добавляем ее и продолжаем рекурсию. Если правых скобок осталось больше, чем левых, то вставляем правую скобку и продолжаем рекурсию.

```
void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
              int count) {
  if (leftRem < 0 || rightRem < leftRem) return; // Некорректное состояние

  if (leftRem == 0 && rightRem == 0) { /* Скобок не осталось */
    String s = String.copyValueOf(str);
    list.add(s);
  } else {
    /* Добавить левую скобку, если они еще остались. */
     if (leftRem > 0) {
       str[count] = '(';
       addParen(list, leftRem - 1, rightRem, str, count + 1);
     }

     /* Добавить правую скобку, если выражение корректно */
     if (rightRem > leftRem) {
```

```
17        str[count] = ')';
18        addParen(list, leftRem, rightRem - 1, str, count + 1);
19      }
20    }
21  }
22
23  ArrayList<String> generateParens(int count) {
24    char[] str = new char[count*2];
25    ArrayList<String> list = new ArrayList<String>();
26    addParen(list, count, count, str, 0);
27    return list;
28  }
```

Поскольку мы добавляем левые и правые скобки для каждого индекса в строке, индексы не повторяются, и каждая строка гарантированно будет уникальной.

8.10. Реализуйте функцию заливки краской, которая поддерживается во многих графических редакторах. Дана плоскость (двумерный массив цветов), точка и цвет, которым нужно заполнить все окружающее пространство, изначально окрашенное в другой цвет.

РЕШЕНИЕ

Для начала наглядно представим, как должен работать метод. Когда мы вызываем `paintFill` (например, щелчком на инструменте заливки в графическом редакторе), — допустим, на зеленом пикселе, — границы заливки должны распространиться наружу. Мы продвигаемся все дальше и дальше, вызывая `paintFill` для окружающих пикселов. Если цвет пиксела отличается от зеленого, мы останавливаемся.

Этот алгоритм может быть реализован рекурсивно:

```
1  enum Color { Black, White, Red, Yellow, Green }
2
3  boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4    if (screen[r][c] == ncolor) return false;
5    return PaintFill(screen, r, c, screen[r][c], ncolor);
6  }
7
8  boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9    if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10     return false;
11   }
12
13   if (screen[r][c] == ocolor) {
14     screen[r][c] = ncolor;
15     PaintFill(screen, r - 1, c, ocolor, ncolor); // Вверх
16     PaintFill(screen, r + 1, c, ocolor, ncolor); // Вниз
17     PaintFill(screen, r, c - 1, ocolor, ncolor); // Влево
18     PaintFill(screen, r, c + 1, ocolor, ncolor); // Вправо
19   }
20   return true;
21 }
```

Когда вы используете имена переменных `x` и `y` в реализации, будьте внимательны с порядком следования `x` и `y` в массиве `screen[y][x]`. Поскольку `x` соответствует

горизонтальному направлению, эта переменная описывает номер столбца, а не номер строки. Значение `y` соответствует номеру строки. В этом месте очень легко допустить ошибку как на собеседовании, так и при повседневном программировании. Обычно намного правильнее использовать имена `row` и `column`, как сделано в нашем примере.

Алгоритм выглядит знакомо? Так и должно быть! Фактически перед вами поиск в глубину на графе. На каждом пикселе алгоритм передвигается наружу к окружающим пикселам. Остановка происходит после полного обхода всех прилегающих пикселов данного цвета.

Также возможна реализация заливки на базе поиска в ширину.

8.11. Дано неограниченное количество монет достоинством 25, 10, 5 и 1 цент. Напишите код, определяющий количество способов представления `n` центов.

РЕШЕНИЕ

Это рекурсивная задача, поэтому давайте разберемся, как вычислить `makeChange(n)` на основе предыдущих решений (подзадач).

Пусть `n = 100`. Мы хотим вычислить количество способов представления 100 центов. Как эта задача связана со своими подзадачами?

Известно, что в представлении 100 центов могут использоваться 0, 1, 2, 3 или 4 четвертака (монеты с номиналом 25 центов):

```
makeChange(100) =
    makeChange(100 с использованием 0 четвертаков) +
    makeChange(100 с использованием 1 четвертака) +
    makeChange(100 с использованием 2 четвертаков) +
    makeChange(100 с использованием 3 четвертаков) +
    makeChange(100 с использованием 4 четвертаков)
```

Двигаемся дальше: попробуем упростить некоторые из этих задач. Например, `makeChange(100 с использованием 1 четвертака) = makeChange(75 с использованием 0 четвертаков)`. Это так, потому что если мы должны использовать один четвертак для представления 100 центов, оставшиеся варианты соответствуют различным представлениям 75 центов.

Та же логика применима для `makeChange(100 с использованием 2 четвертаков)`, `makeChange(100 с использованием 3 четвертаков)` и `makeChange(100 с использованием 4 четвертаков)`. Приведенное ранее выражение можно свести к следующему:

```
makeChange(100) =
    makeChange(100 с использованием 0 четвертаков) +
    makeChange(75 с использованием 0 четвертаков) +
    makeChange(50 с использованием 0 четвертаков) +
    makeChange(25 с использованием 0 четвертаков) +
    1
```

Заметьте, что последнее выражение — `makeChange(100 с использованием 4 четвертаков)` — равно 1.

Что делать дальше? Теперь мы израсходовали все четвертаки и можем использовать следующую самую крупную монету — 10 центов.

Подход, использованный для четвертаков, подойдет и для 10-центовых монет, но мы применим его для *каждой* из четырех частей приведенного выше выражения. Так, для первой части:

```
makeChange(100 с использованием 0 четвертаков) =
    makeChange(100 с использованием 0 четвертаков, 0 монет в 10 центов) +
    makeChange(100 с использованием 0 четвертаков, 1 монеты в 10 центов) +
    makeChange(100 с использованием 0 четвертаков, 2 монет в 10 центов) +
    ...
    makeChange(100 с использованием 0 четвертаков, 10 монет в 10 центов)
makeChange(75 с использованием 0 четвертаков) =
    makeChange(75 с использованием 0 четвертаков, 0 монет в 10 центов) +
    makeChange(75 с использованием 0 четвертаков, 1 монеты в 10 центов) +
    makeChange(75 с использованием 0 четвертаков, 2 монет в 10 центов) +
    ...
    makeChange(75 с использованием 0 четвертаков, 7 монет в 10 центов)
makeChange(50 с использованием 0 четвертаков) =
    makeChange(50 с использованием 0 четвертаков, 0 монет в 10 центов) +
    makeChange(50 с использованием 0 четвертаков, 1 монеты в 10 центов) +
    makeChange(50 с использованием 0 четвертаков, 2 монет в 10 центов) +
    ...
    makeChange(50 с использованием 0 четвертаков, 5 монет в 10 центов)
makeChange(25 с использованием 0 четвертаков) =
    makeChange(25 с использованием 0 четвертаков, 0 монет в 10 центов) +
    makeChange(25 с использованием 0 четвертаков, 1 монеты в 10 центов) +
    makeChange(25 с использованием 0 четвертаков, 2 монет в 10 центов)
```

После этого можно перейти к монеткам в 5 и 1 цент. В результате мы получим древовидную рекурсивную структуру, где каждый вызов расширяется до четырех или больше вызовов.

Базовый случай для нашей рекурсии — полностью сведенное (упрощенное) выражение. Например, `makeChange(50 с использованием 0 четвертаков, 5 монет в 10 центов)` полностью сводится к 1, так как 5 монет по 10 центов дают ровно 50 центов.

Рекурсивный алгоритм будет иметь примерно такой вид:

```
1  int makeChange(int amount, int[] denoms, int index) {
2    if (index >= denoms.length - 1) return 1; // Последний делитель
3    int denomAmount = denoms[index];
4    int ways = 0;
5    for (int i = 0; i * denomAmount <= amount; i++) {
6      int amountRemaining = amount - i * denomAmount;
7      ways += makeChange(amountRemaining, denoms, index + 1);
8    }
9    return ways;
10 }
11
12 int makeChange(int n) {
13   int[] denoms = {25, 10, 5, 1};
14   return makeChange(n, denoms, 0);
15 }
```

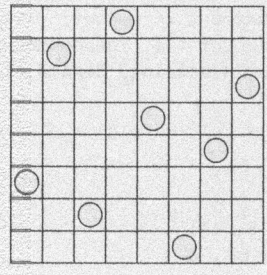
Такое решение работает, но оно не настолько эффективно, как хотелось бы. Проблема в том, что мы будем рекурсивно вызывать `makeChange` для одинаковых значений суммы и индекса.

Проблема решается сохранением ранее вычисленных значений. В кэше должно храниться соответствие между каждой парой (*сумма*, *индекс*) и заранее вычисленным результатом.

```
1  int makeChange(int n) {
2    int[] denoms = {25, 10, 5, 1};
3    int[][] map = new int[n + 1][denoms.length]; // Кэшированные значения
4    return makeChange(n, denoms, 0, map);
5  }
6
7  int makeChange(int amount, int[] denoms, int index, int[][] map) {
8    if (map[amount][index] > 0) { // Выборка значения
9      return map[amount][index];
10   }
11   if (index >= denoms.length - 1) return 1; // Остается один делитель
12   int denomAmount = denoms[index];
13   int ways = 0;
14   for (int i = 0; i * denomAmount <= amount; i++) {
15     // Перейти к следующему делителю для i монет с номиналом denomAmount
16     int amountRemaining = amount - i * denomAmount;
17     ways += makeChange(amountRemaining, denoms, index + 1, map);
18   }
19   map[amount][index] = ways;
20   return ways;
21 }
```

Обратите внимание на использование двумерного целочисленного массива для хранения ранее вычисленных значений. Такой вариант проще, но он требует дополнительных затрат памяти. Также можно воспользоваться хеш-таблицей, отображающей `amount` на новую хеш-таблицу, которая, в свою очередь, выполняет отображение `denom` на заранее вычисленное значение. Также возможны другие альтернативные структуры данных.

8.12. Напишите алгоритм, находящий все варианты расстановки восьми ферзей на шахматной доске размером 8×8 так, чтобы никакие две фигуры не располагались на одной горизонтали, вертикали или диагонали (учитываются не только две главные, но и все остальные диагонали).



РЕШЕНИЕ

Есть 8 ферзей, которые нужно расставить на доске размером 8×8 так, чтобы они не оказались на одной вертикали, горизонтали или диагонали. Мы знаем,

что каждая вертикаль, горизонталь и диагональ должны использоваться только один раз.

Представьте, что последний ферзь был поставлен на 8-й горизонтали (это абсолютно нормальное предположение, ведь порядок размещения ферзей не учитывается). На какой клетке 8-й горизонтали находится ферзь? Существует восемь вариантов, по одному для каждой вертикали.

Так, мы хотим знать все варианты размещения 8 ферзей на доске размером 8×8:

```
Число способов расположить 8 ферзей на доске 8x8 =
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 0) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 1) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 2) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 3) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 4) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 5) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 6) +
    способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 7)
```

Для вычисления каждого слагаемого можно воспользоваться аналогичным способом:

```
Число способов расположить 8 ферзей на доске 8x8 с ферзем на клетке (7, 3) =
    способов ... с ферзями на (7, 3) и (6, 0) +
    способов ... с ферзями на (7, 3) и (6, 1) +
    способов ... с ферзями на (7, 3) и (6, 2) +
    способов ... с ферзями на (7, 3) и (6, 4) +
    способов ... с ферзями на (7, 3) и (6, 5) +
    способов ... с ферзями на (7, 3) и (6, 6) +
    способов ... с ферзями на (7, 3) и (6, 7)
```

Обратите внимание: комбинации с ферзями на клетках (7, 3) и (6, 3) учитывать не нужно, поскольку они нарушают требование, что каждый ферзь должен находиться на собственной вертикали/горизонтали/диагонали.

Теперь решение задачи становится вполне прямолинейным:

```
1  int GRID_SIZE = 8;
2
3  void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
4    if (row == GRID_SIZE) { // Найдено допустимое размещение
5      results.add(columns.clone());
6    } else {
7      for (int col = 0; col < GRID_SIZE; col++) {
8        if (checkValid(columns, row, col)) {
9          columns[row] = col;  // Размещение ферзя
10          placeQueens(row + 1, columns, results);
11       }
12     }
13    }
14  }
15
16  /* Проверить, можно ли разместить ферзя в позиции (row1, column1).
17   * Для этого нужно убедиться в отсутствии ферзей на той же вертикали или
18   * диагонали. Проверять наличие ферзя на той же горизонтали не нужно,
19   * потому что горизонталь заведомо свободна. */
20  boolean checkValid(Integer[] columns, int row1, int column1) {
```

```
21    for (int row2 = 0; row2 < row1; row2++) {
22      int column2 = columns[row2];
23      /* Проверяем, делает ли (row2, column2) позицию (row1, column1)
24       * недействительной для размещения ферзя. */
25
26      /* Проверить наличие ферзя на той же вертикали. */
27      if (column1 == column2) {
28        return false;
29     }
30
31      /* Проверка диагоналей: если расстояние между вертикалями равно
32       * расстоянию между горизонталями, они лежат на одной диагонали. */
33      int columnDistance = Math.abs(column2 - column1);
34
35      /* row1 > row2, в вызове abs нет необходимости */
36      int rowDistance = row1 - row2;
37      if (columnDistance == rowDistance) {
38        return false;
39     }
40    }
41    return true;
42  }
```

Так как на каждой горизонтали может быть только один ферзь, хранить доску в виде полной матрицы 8×8 не нужно. Хватит одномерного массива, где `column [row] = c` указывает, что на горизонтали `r` расположен ферзь, занимающий вертикаль `c`.

8.13. Имеется штабель из n ящиков шириной w_i, высотой h_i и глубиной d_i. Ящики нельзя поворачивать, добавлять ящики можно только на верх штабеля. Каждый нижний ящик в стопке по высоте, ширине и глубине должен быть строго больше ящика, который находится на нем. Реализуйте метод для вычисления высоты самого высокого штабеля (высота штабеля равна сумме высот всех ящиков).

РЕШЕНИЕ

Чтобы решить эту задачу, нужно определить связи между различными подзадачами.

Решение 1

Предположим, имеются ящики $b_1, b_2, \ldots, b_n$. Наибольший штабель, который можно создать со всеми ящиками, равен максимуму из (наибольший стек с основанием b_1, наибольший стек с основанием b_2, ..., наибольший стек с основанием b_n). Таким образом, экспериментируя с построением наибольшего возможного штабеля на базе каждого ящика, мы найдем наибольший возможный штабель.

Но как найти наибольший штабель с конкретным основанием? Да все так же. Нужно экспериментировать с разными ящиками для второго уровня (и далее для каждого уровня).

Конечно, экспериментировать нужно только с возможными вариантами. Если b_5 больше, чем b_1, нет смысла в попытке создать штабель $\{b_1, b_5, \ldots\}$ — ведь уже известно, что b_1 не может находиться под b_5.

Здесь можно провести небольшую оптимизацию. В формулировке задачи сказано, что нижние ящики должны быть строго больше ящиков, расположенных выше, по всем измерениям. Следовательно, если отсортировать (по убыванию) ящики по одному любому измерению, то проверка в обратном направлении по списку становится излишней. Ящик b_1 не может находиться над b_5, потому что его высота (или другое измерение, по которому была выполнена сортировка) больше высоты b_5.

Ниже приведена рекурсивная реализация этого алгоритма.

```
1  int createStack(ArrayList<Box> boxes) {
2    /* Сортировка по убыванию высоты. */
3    Collections.sort(boxes, new BoxComparator());
4    int maxHeight = 0;
5    for (int i = 0; i < boxes.size(); i++) {
6      int height = createStack(boxes, i);
7      maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {
13   Box bottom = boxes.get(bottomIndex);
14   int maxHeight = 0;
15   for (int i = bottomIndex + 1; i < boxes.size(); i++) {
16     if (boxes.get(i).canBeAbove(bottom)) {
17       int height = createStack(boxes, i);
18       maxHeight = Math.max(height, maxHeight);
19     }
20   }
21   maxHeight += bottom.height;
22   return maxHeight;
23 }
24
25 class BoxComparator implements Comparator<Box> {
26   @Override
27   public int compare(Box x, Box y){
28     return y.height - x.height;
29   }
30 }
```

Проблема этого кода в том, что он крайне неэффективен. Мы пытаемся найти лучшее решение вида $\{b_3, b_4, \ldots\}$, даже в том случае, когда нашли лучшее решение с b_4 в основании. Вместо того чтобы генерировать решения с нуля, можно кэшировать результаты, используя метод мемоизации.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int maxHeight = 0;
4    int[] stackMap = new int[boxes.size()];
5    for (int i = 0; i < boxes.size(); i++) {
6      int height = createStack(boxes, i, stackMap);
7      maxHeight = Math.max(maxHeight, height);
8    }
9    return maxHeight;
10 }
11
```

```
12  int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13    if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14      return stackMap[bottomIndex];
15    }
16
17    Box bottom = boxes.get(bottomIndex);
18    int maxHeight = 0;
19    for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20      if (boxes.get(i).canBeAbove(bottom)) {
21        int height = createStack(boxes, i, stackMap);
22        maxHeight = Math.max(height, maxHeight);
23     }
24    }
25    maxHeight += bottom.height;
26    stackMap[bottomIndex] = maxHeight;
27    return maxHeight;
28  }
```

Так как индекс связывается только с высотой, в качестве хеш-таблицы можно с таким же успехом использовать целочисленный массив.

Будьте очень внимательны с тем, что представляет собой каждая ячейка хеш-таблицы. В этом коде `stackMap[i]` представляет самый высокий штабель с ящиком `i` в основании. Прежде чем извлекать значение из хеш-таблицы, необходимо убедиться в том, что ящик `i` может размещаться выше текущего основания.

Решение 2

Также рекурсивный алгоритм можно рассматривать как принятие решения о включении конкретного ящика в штабель. (Ящики снова сортируются по убыванию некоторого измерения, например высоты.)

Сначала мы выбираем, включать ли в штабель ящик 0. Берем один рекурсивный путь с ящиком 0 в основании и один рекурсивный путь без ящика 0; возвращаем лучший из двух вариантов.

Далее выбираем, включать ли в штабель ящик 1. Берем один рекурсивный путь с ящиком 1 в основании и один рекурсивный путь без ящика 1; возвращаем лучший из двух вариантов.

Как и прежде, для сохранения максимальной высоты штабеля с конкретным ящиком в основании применяется мемоизация.

```
1  int createStack(ArrayList<Box> boxes) {
2    Collections.sort(boxes, new BoxComparator());
3    int[] stackMap = new int[boxes.size()];
4    return createStack(boxes, null, 0, stackMap);
5  }
6
7  int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8    if (offset >= boxes.size()) return 0; // Базовый случай
9
10   /* Высота с заданным основанием */
11   Box newBottom = boxes.get(offset);
12   int heightWithBottom = 0;
13   if (bottom == null || newBottom.canBeAbove(bottom)) {
14     if (stackMap[offset] == 0) {
```

```
15        stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16        stackMap[offset] += newBottom.height;
17      }
18       heightWithBottom = stackMap[offset];
19    }
20
21    /* Без основания */
22    int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24    /* Вернуть лучший из двух вариантов. */
25    return Math.max(heightWithBottom, heightWithoutBottom);
26  }
```

И снова будьте внимательны при чтении и вставке значений в хеш-таблицу. Обычно такие операции стоит выполнять симметрично, как в строках 15 и 16–18.

8.14. Дано логическое выражение, построенное из символов `0`, `1`, `&`, `|` и `^`, и нужное логическое значение `result`. Напишите функцию, подсчитывающую количество вариантов расстановки в логическом выражении круглых скобок, для которых результат выражения равен `result`.

Пример:

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

РЕШЕНИЕ

Как в других рекурсивных задачах, ключ к решению — выяснить связь между задачей и подзадачами.

Метод «грубой силы»

Рассмотрим выражение вида `0^0&0^1|1` и целевой результат `true`. Как разбить `countEval(0^0&0^1|1, true)` на меньшие подзадачи?

По сути? нужно перебрать все возможные места для расстановки круглых скобок.

```
countEval(0^0&0^1|1, true) =
   countEval(0^0&0^1|1 со скобками вокруг символа 1, true)
 + countEval(0^0&0^1|1 со скобками вокруг символа 3, true)
 + countEval(0^0&0^1|1 со скобками вокруг символа 5, true)
 + countEval(0^0&0^1|1 со скобками вокруг символа 7, true)
```

Что дальше? Рассмотрим одно из этих выражений: круглые скобки вокруг символа 3. Получаем `(0^0)&(0^1)`.

Чтобы выражение было истинным, и левая и правая стороны должны быть истинными. Следовательно:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

Причина, по которой мы перемножаем результаты левой и правой стороны, заключается в том, что каждый результат с каждой из сторон образует пару с результатом

с другой стороны, формируя уникальную комбинацию. Каждая составляющая раскладывается на меньшие подзадачи аналогичным образом.

Что произойдет, если используется операция "|" (OR)? Или "^" (XOR)?

Если используется операция OR, то истинной должна быть левая или правая сторона или обе одновременно.

```
countEval(left | right, true) =
   + countEval(left, true)  *  countEval(right, false)
   + countEval(left, false) *  countEval(right, true)
   + countEval(left, true)  *  countEval(right, true)
```

Если используется операция XOR, то истинной может быть либо левая, либо правая сторона, но не обе сразу.

```
countEval(left ^ right, true) =
   + countEval(left, true)  *  countEval(right, false)
   + countEval(left, false) *  countEval(right, true)
```

А если бы мы стремились прийти к результату false? Описанная логика заменяется противоположной:

```
countEval(left & right, false) =
     countEval(left, true)  * countEval(right, false)
   + countEval(left, false) * countEval(right, true)
   + countEval(left, false) * countEval(right, false)
countEval(left | right, false) =
   + countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) =
   + countEval(left, false) * countEval(right, false)
   + countEval(left, true)  * countEval(right, true)
```

Также можно воспользоваться логикой, описанной выше, и вычесть результат из общего количества способов вычисления выражения:

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

Код становится чуть более компактным.

```
1  int countEval(String s, boolean result) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4
5    int ways = 0;
6    for (int i = 1; i < s.length(); i += 2) {
7      char c = s.charAt(i);
8      String left = s.substring(0, i);
9      String right = s.substring(i + 1, s.length());
10
11      /* Вычислить каждую сторону для каждого результата. */
12      int leftTrue = countEval(left, true);
13      int leftFalse = countEval(left, false);
14      int rightTrue = countEval(right, true);
15      int rightFalse = countEval(right, false);
```

```
16      int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18      int totalTrue = 0;
19      if (c == '^') { // Требование: true и false
20        totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21      } else if (c == '&') { // Требование: два true
22        totalTrue = leftTrue * rightTrue;
23      } else if (c == '|') { // Требование: что угодно, кроме двух false
24        totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                    leftTrue * rightFalse;
26     }
27
28      int subWays = result ? totalTrue : total - totalTrue;
29      ways += subWays;
30    }
31
32    return ways;
33  }
34
35  boolean stringToBool(String c) {
36    return c.equals("1") ? true : false;
37  }
```

За вычисление результатов `false` на базе результатов `true` и непосредственное вычисление значений `{leftTrue, rightTrue, leftFalse, rightFalse}` приходится расплачиваться дополнительной работой в некоторых случаях. Например, если мы ищем варианты, при которых результат операции AND (`&`) будет равен `true`, результаты `leftFalse` и `rightFalse` вообще не понадобятся. Аналогичным образом, если мы ищем варианты, при которых результат операции OR (`|`) будет равен `false`, не понадобятся результаты `leftTrue` и `rightTrue`.

Текущая верия кода не учитывает, что нужно, а что не нужно, и просто вычисляет все значения. Вероятно, это разумный компромисс (особенно с учетом ограничений «программирования у доски»), так как код становится заметно короче и писать его будет менее утомительно. Какой бы подход вы ни выбрали, обсудите его достоинства и недостатки с интервьюером.

Впрочем, существуют и другие, более важные оптимизации.

Оптимизированные решения

Если пойти по рекурсивному пути, можно заметить, что одни и те же вычисления выполняются повторно.

Рассмотрим выражение `0^0&0^1|1` и следующие пути рекурсии:

- Добавить скобки вокруг символа 1. `(0)^(0&0^1|1)`
  - Добавить скобки вокруг символа 3. `(0)^((0)&(0^1|1))`
- Добавить скобки вокруг символа 3. `(0^0)&(0^1|1)`
  - Добавить скобки вокруг символа 1. `((0)^(0))&(0^1|1)`

И хотя эти два выражения различны, у них есть общий компонент: `(0^1|1)`. Следует использовать результаты уже выполненной работы, а для этого можно воспользоваться мемоизацией: нужно лишь сохранить результат `countEval(выражение`,

результат) для всех вариантов выражения и результата. Встретив выражение, которое вычислялось ранее, мы просто возвращаем его из кэша.

```
1  int countEval(String s, boolean result, HashMap<String, Integer> memo) {
2    if (s.length() == 0) return 0;
3    if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4    if (memo.containsKey(result + s)) return memo.get(result + s);
5
6    int ways = 0;
7
8    for (int i = 1; i < s.length(); i += 2) {
9      char c = s.charAt(i);
10     String left = s.substring(0, i);
11     String right = s.substring(i + 1, s.length());
12     int leftTrue = countEval(left, true, memo);
13     int leftFalse = countEval(left, false, memo);
14     int rightTrue = countEval(right, true, memo);
15     int rightFalse = countEval(right, false, memo);
16     int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18     int totalTrue = 0;
19     if (c == '^') {
20       totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21     } else if (c == '&') {
22       totalTrue = leftTrue * rightTrue;
23     } else if (c == '|') {
24       totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                   leftTrue * rightFalse;
26    }
27
28     int subWays = result ? totalTrue : total - totalTrue;
29     ways += subWays;
30   }
31
32   memo.put(result + s, ways);
33   return ways;
34 }
```

К дополнительным преимуществам этого кода относится возможность использования одной подстроки в разных частях выражения. Например, выражение вида `0^1^0&0^1^0` содержит два вхождения `0^1^0`. Кэширование результата подстроки в таблице мемоизации позволит повторно использовать в правой части выражения результат, вычисленный для левой части.

9

Масштабируемость и проектирование систем

9.1. Представьте, что вы создаете службу, которая получает и обрабатывает простейшую информацию от 1000 клиентских приложений о курсе акций в конце торгового дня (открытие, закрытие, максимум, минимум). Предположим, что все данные получены, и вы можете сами выбрать формат для их хранения. Как должна выглядеть служба, предоставляющая информацию клиентским приложениям? Вы должны обеспечить развертывание, продвижение, мониторинг и обслуживание системы. Опишите различные варианты построения службы, которые вы рассмотрели, и обоснуйте свой подход. Вы можете использовать любые технологии и любой механизм предоставления информации клиентским приложениям.

РЕШЕНИЕ

Из постановки задачи понятно, что мы должны сосредоточиться на предоставлении информации клиентам. Будем считать, что существуют какие-то сценарии, которые по волшебству собирают нужную информацию.

Начнем со списка аспектов, которые необходимо учитывать при проектировании:

- *Удобство для клиента* — служба должна была простой в реализации и полезной для клиента.
- *Удобство для производителя* — сервис должен быть максимально прост в реализации и не создавать лишней работы. Нужно учитывать не только расходы на его разработку, но и стоимость обслуживания.
- *Гибкость (учет будущих потребностей)* — задача сформулирована в стиле «Как бы вы поступили в реальном проекте», поэтому и относиться к ней следует как к реальному проекту. В идеале нам хотелось бы сохранить максимум гибкости на случай изменения требований или ситуации.
- *Масштабируемость и эффективность* — нельзя забывать об эффективности решения, чтобы избежать лишних нагрузок на службу.

Учитывая все сказанное, можно сформулировать несколько предложений.

Предложение 1

Данные можно хранить в обычных текстовых файлах, которые клиенты смогут загружать через некий FTP-сервер. Такая система проста в сопровождении, так как она позволяет легко просматривать файлы, создавать резервные копии, но это потребует

сложного разбора данных. К тому же если в текстовый файл будут добавлены новые данные, они могут нарушить работу механизма разбора клиентских данных.

Предложение 2

Можем использовать стандартную базу данных SQL, к которой клиенты будут подключаться напрямую. Такое решение предоставляет следующие преимущества:

- Упрощение обработки запросов, особенно в случаях, когда нужно предоставить расширенные возможности. Например, оно позволяет легко и эффективно выполнить запрос типа «выберите все акции, имеющие цену открытия больше N и цену закрытия меньше M».
- Откат, резервное копирование и средства безопасности могут быть реализованы стандартными средствами баз данных. Нам не придется изобретать велосипед.
- Относительная простота интеграции клиентов с существующими приложениями. SQL-интеграция — стандартная функция во многих средах разработки приложений (SDE).

А что с недостатками?

- SQL-база данных предоставляет намного больше возможностей, нежели реально необходимо. Хранение небольшого количества байт информации не требует сложной подсистемы SQL.
- Человек не может «читать» базу данных напрямую, поэтому нам придется создать дополнительный уровень просмотра и сопровождения данных. Это увеличит расходы на реализацию проекта.
- Безопасность: хотя базы данных SQL хорошо защищены, необходимо следить за тем, чтобы клиент не получил прав доступа, без которых он может обойтись. Даже если клиенты не собираются выполнять какие-то злонамеренные действия, они могут попытаться выполнить затратные и неэффективные запросы, создавая лишнюю нагрузку на сервер.

Эти недостатки не означают, что от баз данных SQL нужно отказаться. Просто необходимо хорошо понимать недостатки этой технологии.

Предложение 3

XML — другой отличный способ распространения информации. Наши данные имеют фиксированный формат и размер: `company_name`, `open`, `high`, `low`, `closing price`. Разметка XML может выглядеть примерно так:

```
<root>
  <date value="2008-10-12">
    <company name="foo">
    <open>126.23</open>
      <high>130.27</high>
    <low>122.83</low>
      <closingPrice>127.30</closingPrice>
   </company>
    <company name="bar">
     <open>52.73</open>
       <high>60.27</high>
```

```
12        <low>50.29</low>
13      <closingPrice>54.91</closingPrice>
14     </company>
15    </date>
16     <date value="2008-10-11"> . . . </date>
17  </root>
```

Преимущества этого подхода очевидны:

- Простота распространения и чтения (как для компьютеров, так и для людей). Именно поэтому формат XML считается стандартной моделью совместного использования и распределения данных.
- В большинстве языков программирования существует библиотека для разбора данных в формате XML, так что решение достаточно легко реализуется на стороне клиента.
- Всегда можно вставить новые данные в XML-файл, просто добавив дополнительные узлы. Это не помешает работе клиентского парсера (при условии его нормальной реализации).
- Так как данные хранятся в XML-файле, можно использовать существующие инструменты резервного копирования. Нам не придется реализовать собственные средства архивации.

Недостатки:

- Это решение передает клиентам всю информацию, даже если им нужна ее малая часть. В этом отношении оно неэффективно.
- Выполнение любых запросов к данным потребует разбора всего файла.

Независимо от метода, который будет использоваться для хранения данных, можно предоставить клиентам веб-службу (например, на базе SOAP) для работы с данными. Это добавляет дополнительный «слой» к нашей работе, но позволяет гарантировать безопасность и облегчить интеграцию системы.

С другой стороны, клиенты будут получать только те данные, которые мы захотим им предоставить, — и это может быть как достоинством, так и недостатком. «Чистая» реализация на базе SQL позволяет выполнить любой запрос клиента (например, выбрать акции с самой высокой ценой), даже если мы изначально не предполагали, что им это понадобится.

Так какой вариант использовать? Здесь нет однозначного ответа. Вероятно, использование текстовых файлов — самый неудачный выбор, но вы можете привести убедительные аргументы в пользу SQL- или XML-решений, с веб-службой или без нее.

Цель подобных вопросов — не получить «правильный» ответ (ибо его не существует). Интервьюер хочет увидеть, как вы проектируете систему и оцениваете компромиссы.

9.2. Какие структуры данных вы бы стали использовать в очень больших социальных сетях вроде Facebook или LinkedIn? Опишите, как вы будете разрабатывать алгоритм для поиска кратчайшей цепочки знакомств между двумя людьми (Я → Боб → Сьюзи → Джейсон → Ты).

РЕШЕНИЕ

Хороший подход к решению этой задачи — снять ограничения и сначала разобраться с упрощенной версией.

Шаг 1. Упрощаем задачу — забудьте о миллионах пользователей

Прежде всего, давайте забудем, что имеем дело с миллионами пользователей. Найдем решение для простого случая.

Можно создать граф и рассматривать каждого человека как узел, а существование связи между двумя узлами говорит, что пользователи — друзья.

Если бы мне нужно было найти связь между людьми, я бы начала с одного человека и просто осуществила поиск в ширину.

Почему не в глубину? Во-первых, поиск у глубину просто найдет путь, который не обязательно окажется кратчайшим. Во-вторых, даже если бы нам был нужен *любой* путь, это будет крайне неэффективно. Два пользователя могут находиться по соседству, но нам придется просмотреть миллионы узлов в их поддеревьях, прежде чем эта (относительно близкая) связь обнаружится.

Также можно провести так называемый *двусторонний поиск в ширину*. Под этим термином понимается выполнение двух поисков в ширину: от начального и целевого узла. Если два поиска встречаются, мы знаем, что путь обнаружен.

В приведенной ниже реализации используются два вспомогательных класса. Класс `BFSData` хранит данные, необходимые для поиска в ширину (например, хеш-таблица `isVisited` и очередь `toVisit`). Класс `PathNode` представляет путь в процессе поиска, с сохранением каждого объекта `Person` и предыдущего узла `previousNode`, посещенного на этом пути.

```
1  LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
2                                   int destination) {
3    BFSData sourceData = new BFSData(people.get(source));
4    BFSData destData = new BFSData(people.get(destination));
5
6    while (!sourceData.isFinished() && !destData.isFinished()) {
7      /* Поиск от исходного узла. */
8      Person collision = searchLevel(people, sourceData, destData);
9      if (collision != null) {
10       return mergePaths(sourceData, destData, collision.getID());
11     }
12
13     /* Поиск от конечного узла. */
14     collision = searchLevel(people, destData, sourceData);
15     if (collision != null) {
16       return mergePaths(sourceData, destData, collision.getID());
17     }
18   }
19   return null;
20 }
21
22 /* Проведение поиска на одном уровне. */
23 Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
24                    BFSData secondary) {
25   /* Поиск проводится только на одном уровне. Подсчитать количество
```

```
26      * узлов на уровне primary и обработать такое количество узлов.
27      * Мы продолжим добавлять узлы в конце. */
28     int count = primary.toVisit.size();
29     for (int i = 0; i < count; i++) {
30       /* Извлечение первого узла. */
31       PathNode pathNode = primary.toVisit.poll();
32       int personId = pathNode.getPerson().getID();
33
34       /* Проверить, посещался ли он ранее. */
35       if (secondary.visited.containsKey(personId)) {
36         return pathNode.getPerson();
37      }
38
39       /* Добавление друзей в очередь. */
40       Person person = pathNode.getPerson();
41       ArrayList<Integer> friends = person.getFriends();
42       for (int friendId : friends) {
43         if (!primary.visited.containsKey(friendId)) {
44           Person friend = people.get(friendId);
45           PathNode next = new PathNode(friend, pathNode);
46           primary.visited.put(friendId, next);
47           primary.toVisit.add(next);
48       }
49      }
50     }
51     return null;
52   }
53
54   /* Слияние путей при соприкосновении путей поиска. */
55   LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
56     PathNode end1 = bfs1.visited.get(connection); // end1 -> source
57     PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
58     LinkedList<Person> pathOne = end1.collapse(false);
59      LinkedList<Person> pathTwo = end2.collapse(true); // Обратный путь
60     pathTwo.removeFirst(); // удалить соединение
61     pathOne.addAll(pathTwo); // добавить второй путь
62     return pathOne;
63   }
64
65   class PathNode {
66     private Person person = null;
67     private PathNode previousNode = null;
68     public PathNode(Person p, PathNode previous) {
69       person = p;
70       previousNode = previous;
71     }
72
73     public Person getPerson() { return person; }
74
75     public LinkedList<Person> collapse(boolean startsWithRoot) {
76       LinkedList<Person> path = new LinkedList<Person>();
77       PathNode node = this;
78       while (node != null) {
79         if (startsWithRoot) {
80           path.addLast(node.person);
81         } else {
82           path.addFirst(node.person);
```

```
83      }
84        node = node.previousNode;
85     }
86      return path;
87    }
88  }
89
90  class BFSData {
91    public Queue<PathNode> toVisit = new LinkedList<PathNode>();
92    public HashMap<Integer, PathNode> visited =
93      new HashMap<Integer, PathNode>();
94
95    public BFSData(Person root) {
96      PathNode sourcePath = new PathNode(root, null);
97      toVisit.add(sourcePath);
98      visited.put(root.getID(), sourcePath);
99    }
100
101    public boolean isFinished() {
102      return toVisit.isEmpty();
103    }
104  }
```

Вас удивляет то, что это решение работает быстрее? Следующие вычисления помогут вам понять, почему это так.

Допустим, у каждого человека `k` друзей, а у узлов `S` и `D` имеется общий друг `C`.

- Традиционный поиск в ширину от `S` к `D`: посещаются приблизительно `k+k*k` узлов: каждый из `k` друзей `S`, а затем каждый из их `k` друзей.
- Двусторонний поиск в ширину: посещаются `2k` узлов: каждый из `k` друзей `S` и каждый из `k` друзей `D`.

Конечно, `2k` намного меньше `k+k*k`.

Обобщая эти рассуждения для пути длины `q`, получаем:

- Традиционный поиск в ширину: $O(kq)$.
- Двусторонний поиск в ширину: $O(kq/2+kq/2)$, что эквивалентно $O(kq/2)$.

Если представить путь вида `A->B->C->D->E`, в котором у каждого человека по 100 друзей, разница будет весьма значительной. Традиционный поиск в ширину потребует проверки 100 миллионов (100^4) узлов, тогда как для двустороннего поиска в ширину будет достаточно всего 20 000 узлов (2×100^2).

Двусторонний поиск в глубину в общем случае работает быстрее традиционного варианта. С другой стороны, для него должны быть доступны оба узла (начальный и конечный), а это условие выполняется не всегда.

Шаг 2. Возвращаемся к миллионам пользователей

Имея дело с огромными объемами информации LinkedIn или Facebook, невозможно хранить все данные на одном компьютере. Это означает, что простая структура данных `Person` не будет работать, — наши друзья могут оказаться на разных компьютерах. Списки друзей придется заменить списками их идентификаторов и работать с ними следующим образом:

1. Для каждого идентификатора друга: `int machine_index=getMachineIDForUser(personID)`.
2. Переходим на компьютер `#machine_index`.
3. На этом компьютере делаем: `Person friend = getPersonWithID(person_id)`.

Приведенный далее код демонстрирует этот процесс. Мы определили класс `Server`, хранящий список всех компьютеров, и класс `Machine`, представляющий отдельную машину. В обоих классах имеются хеш-таблицы, обеспечивающие эффективный поиск данных.

```
class Server {
  HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
  HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer,
Integer>();

  public Machine getMachineWithId(int machineID) {
    return machines.get(machineID);
  }

  public int getMachineIDForUser(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     return machineID == null ? -1 : machineID;
   }

   public Person getPersonWithID(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     if (machineID == null) return null;

     Machine machine = getMachineWithId(machineID);
     if (machine == null) return null;

     return machine.getPersonWithID(personID);
   }
 }

 class Person {
   private ArrayList<Integer> friends = new ArrayList<Integer>();
   private int personID;
   private String info;

   public Person(int id) { this.personID = id; }
   public String getInfo() { return info; }
   public void setInfo(String info) { this.info = info; }
   public ArrayList<Integer> getFriends() { return friends; }
   public int getID() { return personID; }
   public void addFriend(int id) { friends.add(id); }
 }
```

Возможных направлений оптимизации и сопроводительных вопросов довольно много; ниже рассматриваются лишь некоторые возможности.

Оптимизация. Сокращение количества переходов между компьютерами

Переход от одной машины к другой обходится дорого (с точки зрения системных ресурсов). Вместо того чтобы хаотично перемещаться от машины к машине для каждого нового друга, попробуйте сгруппировать переходы: например, если данные пяти друзей хранятся на одной машине, получите их одновременно.

Оптимизация. Разумное «деление» людей и машин

Чаще всего друзья живут в одной и той же стране. Вместо того чтобы делить данные о пользователях по произвольному принципу, попытайтесь группировать их по стране, городу и т. д. Это сократит количество переходов между машинами.

Вопрос. При поиске в ширину необходимо помечать посещенные узлы. Как бы вы это сделали?

Обычно при поиске в ширину для посещенных узлов устанавливается флаг `visited`, который сохраняется в классе узла. В нашем случае так поступать нельзя. Поскольку одновременно выполняется множество запросов, такое решение помешает редактировать данные.

Вместо этого можно смоделировать пометку узлов с помощью хеш-таблицы, в которой по идентификатору узла можно определить, посещался данный узел или нет.

Другие сопроводительные вопросы:

- В реальном мире на серверах случаются сбои. Как это повлияет на проект?
- Как можно использовать кэширование?
- Ведется ли поиск до конца графа? (Граф может быть бесконечным.) Как решить, когда нужно остановиться?
- Некоторые люди имеют больше друзей, чем другие, следовательно, более вероятно, что таким образом можно найти связь между вами и кем-то еще. Как использовать эти данные для выбора места, с которого начинается обход графа?

Это всего лишь малая часть вопросов, которые вам могут задать на собеседовании.

9.3. Представьте, что вы разрабатываете поискового робота. Как избежать зацикливаний при поиске?

РЕШЕНИЕ

Прежде всего следует задать себе вопрос: при каких условиях в этой задаче может возникнуть бесконечный цикл? Такая ситуация вполне вероятна, например, если мы рассматриваем Всемирную паутину как граф ссылок.

Чтобы предотвратить зацикливание, нужно его обнаружить. Один из способов — создание хеш-таблицы, в которой после посещения страницы `v` устанавливается `hash[v] = true`.

Подобное решение применимо при поиске в ширину. Каждый раз при посещении страницы мы собираем все ее ссылки и добавляем их в конец очереди. Если страница уже посещалась, она просто игнорируется.

Прекрасно, но что означает «страница v уже посещалась»? Что определяет страницу v: ее содержимое или URL-адрес?

Если для идентификации страницы использовать URL, то нужно сознавать, что параметры URL-адреса могут указывать на другую страницу. Например, страница `www.careercup.com/page?id=microsoft-interview-questions` отличается от страницы `www.careercup.com/page?id=google-interview-questions`. С другой стороны, можно добавить параметры без изменения страницы (при условии, что параметр не распознается и не обрабатывается веб-приложением). Например, страница `www.careercup.com?foobar=hello` — это та же страница, что и `www.careercup.com`.

Вы скажете: «Хорошо, давайте идентифицировать страницы на основании их содержимого». Это звучит правильно, но не очень хорошо работает. Предположим, что на домашней странице `careercup.com` представлен некий генерирующийся случайным образом контент. Каждый раз, когда вы посещаете страницу, контент будет другим. Такие страницы можно назвать разными? Нет.

На самом деле идеального способа идентификации страниц не существует, что порядком усложняет задачу.

Один из способов решения — ввести критерий оценки подобия страницы. Если страница похожа на другую страницу, то мы *понижаем приоритет* обхода ее дочерних элементов. Для каждой страницы создается своего рода цифровая подпись, основанная на фрагментах контента и URL-адресе.

Посмотрим, как может работать такой алгоритм.

Существует база данных со списком элементов, которые необходимо обойти. При каждой итерации выбирается страница с самым высоким приоритетом, после чего выполняются следующие действия:

1. Открыть страницу и создать подпись страницы, основанную на определенных секциях страницы и ее URL-адресе.
2. Выдать запрос к базе данных, чтобы узнать, посещалась ли в последнее время страница с этой подписью.
3. Если элемент с такой подписью недавно посещался, страница снова вставляется в базу данных с минимальным приоритетом.
4. Если элемент новый, обработать страницу и добавить ее ссылки в базу данных.

Такой алгоритм не позволит нам полностью обойти Всемирную паутину, но предотвратит зацикливание. Если нам понадобится возможность полного обхода страницы (конечно, это возможно только для меньших систем — например, для интрасетей), можно задать минимальный приоритет, необходимый для проверки страницы.

Это упрощенное решение, но есть множество других. Обычно такая задача переходит в беседу с интервьюером, которая может пойти по разным путям. Например, один из таких путей рассматривается в следующей задаче.

9.4. Имеются 10 миллиардов URL-адресов. Как обнаружить все дублирующиеся документы? Дубликатом в данном случае считается совпадение URL-адресов.

РЕШЕНИЕ

Сколько пространства понадобится для хранения 10 миллиардов URL-адресов? Если в среднем URL-адрес состоит из 100 символов, а каждый символ представляется 4 байтами, то для хранения списка из 10 миллиардов URL понадобится около 4 Тбайт. Скорее всего, хранить так много информации в памяти нереально.

Но для начала представим, что все данные каким-то чудом уместились в памяти — полезно начать с решения упрощенной версии задачи. В этом случае можно создать хеш-таблицу, где каждому дублирующемуся URL-адресу, уже обнаруженному в другой позиции списка, ставится в соответствие значение `true` (альтернативное решение: можно просто отсортировать список и найти дубликаты. Это займет лишнее время без каких-либо реальных преимуществ).

Итак, решение для упрощенной версии задачи найдено. А если объем данных составляет 4000 Гбайт данных, которые нельзя хранить в памяти полностью? Два возможных решения — хранение некоторой части данных на диске или разделение данных между компьютерами.

Решение 1. Хранение данных на диске

Если все данные будут храниться на одной машине, документ будет обрабатываться в два прохода. На первом проходе список разделяется на 4000 фрагментов по 1 Гбайт в каждом. Простой способ — сохранить каждый URL-адрес `u` в файле `<x>.txt`, где `x = hash(u) % 4000`. Таким образом происходит разбивка URL-адресов по хеш-кодам. Все URL-адреса с одинаковым хешем окажутся в одном файле.

На втором проходе фактически реализуется предложенное выше решение: загрузить файл в память, создать хеш-таблицу URL-адресов и найти дубликаты.

Решение 2. Много компьютеров

Этот алгоритм очень похож на предыдущий, но для хранения данных используются разные компьютеры. Вместо того чтобы хранить данные в файле `<x>.txt`, мы отправляем их на машину `x`.

У данного решения есть как преимущества, так и недостатки.

Основное преимущество заключается в том, что можно организовать параллельную работу так, чтобы все 4000 блоков обрабатывались одновременно. Для больших объемов данных мы получаем больший выигрыш во времени.

Недостаток заключается в том, что все 400 машин должны работать без сбоев, что на практике (особенно с большими объемами данных и множеством компьютеров) не всегда получается. Поэтому необходимо предусмотреть обработку отказов. Кроме того, участие такого количества компьютеров заметно повышает сложность системы.

Впрочем, оба решения хороши, и оба требуют обсуждения с интервьюером.

9.5. Представьте веб-сервер упрощенной поисковой системы. В системе есть 100 компьютеров, обрабатывающих поисковые запросы, которые могут генерировать запрос `processSearch(string query)` к другому кластеру компьютеров для получения результата. Компьютер, отвечающий на запрос, выбирается

случайным образом; нет гарантий, что один запрос всегда будет передаваться одной и той же машине. Метод processSearch очень затратный. Разработайте механизм кэширования новых запросов. Объясните, как будет обновляться кэш при изменении данных.

РЕШЕНИЕ

Прежде чем браться за проектирование системы, нужно понять смысл вопроса. Многие из деталей несколько неоднозначны, как и должно быть в подобных вопросах. Мы сделаем разумные предположения, но вы должны обсудить все подробности с интервьюером.

Предположения

Ниже перечислены предположения, которые мы использовали в своем решении. Если вы будете использовать другой подход или другую структуру системы, то и предположения окажутся другими. Некоторые подходы могут быть чуть лучше других, но единственно правильного решения не существует.

- Вся обработка запроса происходит на исходной машине (кроме обращения к `processSearch` при необходимости).
- Количество запросов, которые нужно кэшировать, чрезвычайно велико (миллионы).
- Вызовы между машинами выполняются относительно быстро.
- Результатом запроса является упорядоченный список URL-адресов, в котором каждому адресу соответствует 50 символов заголовка и 200 символов сводки.
- Некоторые запросы настолько популярны, что они *всегда* будут оставаться в кэше.

Еще раз напоминаю, что данные предположения не *единственно* допустимые. Это всего лишь один из возможных вариантов разумных предположений.

Системные требования

При проектировании кэша нужно исходить из того, что мы должны поддерживать две первичные функции.

- Эффективный поиск по заданному ключу;
- Замена устаревших данных новыми.

Нельзя забывать про обновление и очистку кэша, когда результаты запроса меняются. Некоторые запросы очень популярны, поэтому могут постоянно находиться в кэше, а значит, мы не можем ждать, пока содержимое кэша устареет естественным образом.

Шаг 1. Проектирование кэша для одной системы

Решение этой задачи хорошо начать с разработки кэша для одной машины. Как бы вы создали структуру данных, позволяющую легко уничтожать старые данные и производить эффективный поиск по заданному ключу?

- Связный список позволяет легко исключать старые данные: «свежие» элементы находятся в начале. Например, можно удалить последний элемент связного списка, когда список достигнет заданного размера.
- Хеш-таблица позволяет эффективно искать данные, но удаление данных несколько затруднено.

Объединим эти две структуры, чтобы получить максимум преимуществ.

Как и в первом случае, создается связный список, в котором узел перемещается вверх при каждом обращении к нему. Таким образом, в конце списка всегда находится самая неактуальная информация.

Кроме того, мы будем использовать хеш-таблицу, которая устанавливает соответствие между запросом и соответствующим узлом в связанном списке. Это позволит не только эффективно возвращать кэшируемые результаты, но и перемещать узел в начало списка, чтобы поддерживать его в актуальном виде.

Приведенный далее сокращенный вариант кода демонстрирует работу алгоритма. Полная версия этого кода содержится в архиве примеров. Впрочем, на собеседовании от вас вряд ли потребуют написать полный код или представить полностью спроектированный вариант такой системы.

```
1  public class Cache {
2    public static int MAX_SIZE = 10;
3    public Node head, tail;
4    public HashMap<String, Node> map;
5    public int size = 0;
6
7    public Cache() {
8      map = new HashMap<String, Node>();
9    }
10
11    /* Перемещение узла в начало связного списка */
12    public void moveToFront(Node node) { ... }
13    public void moveToFront(String query) { ...   }
14
15    /* Удаление узла из связного списка */
16    public void removeFromLinkedList(Node node) { ... }
17
18    /* Получение результатов из кэша и обновление связного списка */
19    public String[] getResults(String query) {
20      if (!map.containsKey(query)) return null;
21
22      Node node = map.get(query);
23      moveToFront(node); // Обновление актуальности
24      return node.results;
25    }
26
27    /* Вставка результатов в связный список и хеш */
28    public void insertResults(String query, String[] results) {
29      if (map.containsKey(query)) { // Обновление значений
30        Node node = map.get(query);
31        node.results = results;
32        moveToFront(node); // Обновление актуальности
33        return;
34     }
35
```

```
36      Node node = new Node(query, results);
37      moveToFront(node);
38      map.put(query, node);
39
40      if (size > MAX_SIZE) {
41        map.remove(tail.query);
42        removeFromLinkedList(tail);
43     }
44    }
45  }
```

Шаг 2. Переход к нескольким компьютерам

Теперь мы примерно представляем, как решить задачу для одной машины, и можно перейти к проектированию системы с отправкой запросов на множество разных компьютеров. Вспомните условие задачи: нет никакой гарантии, что конкретный запрос будет постоянно попадать на один и тот же компьютер.

Первое, в чем нужно разобраться, — как организовать совместное использование кэша многими машинами. Нужно рассмотреть несколько вариантов.

Вариант 1. У каждой машины есть собственный кэш

Самый простой случай — у каждой машины есть собственный кэш. Тогда, если дважды отправить запрос `"foo"` машине № 1, то повторный запрос обработается гораздо быстрее (результат просто будет прочитан из кэша). Но если запрос `"foo"` сначала попал к машине № 1, а потом к машине № 2, то запросы будут рассматриваться как два разных запроса.

Такое решение работает относительно быстро, так как в нем не используются межмашинные вызовы. К сожалению, эффективность кэширования снижается из-за того, что многие повторные запросы будут обработаны как новые.

Вариант 2. У каждой машины есть копия кэша

Другая крайность — хранение на каждой машине полной копии кэша. Когда в кэш добавляются новые элементы, они передаются всем компьютерам. Вся структура данных — связный список и хеш-таблица — будет дублироваться.

В этом случае общие запросы почти всегда оказываются в кэше, поскольку кэш везде одинаковый. Основной недостаток — обновление кэша означает пересылку данных на `N` машин, где `N` — размер кластера ответа. Кроме того, поскольку каждый элемент кэша должен дублироваться `N` раз, сам кэш будет содержать намного меньше данных.

Вариант 3. У каждой машины есть собственный сегмент данных

Третий вариант — разделение кэша между машинами. Когда компьютеру `i` потребуются результаты запроса, он должен выяснить, какая машина содержит нужное значение, а затем приказать этой машине (`j`), чтобы она нашла запрос в кэше `j`.

Но как узнать, какая машина содержит нужную часть хеш-таблицы?

Один из вариантов — назначать запросы по формуле `hash(query) % N`. Тогда компьютер `i` при помощи этой формулы сможет узнать, что результаты запроса хранятся на машине `j`.

При получении нового запроса к машине `i` она использует формулу и обращается к машине `j`. Машина `j` возвращает значение из своего кэша или вызывает `processSearch(query)` для поиска результатов. После этого машина `j` обновляет свой кэш и возвращает результаты машине `i`.

Еще один вариант — разработать систему так, чтобы машина `j` просто возвращала `null`, если в ее кэше нет этого запроса. После этого машина `i` сама вызовет `processSearch(query)` и отправит результаты машине `j` для хранения. Однако данная реализация увеличивает количество межмашинных обращений без каких-либо реальных преимуществ.

Шаг 3. Обновление результатов при изменении контента

Некоторые запросы могут быть настолько популярны, что при условии достаточно большого кэша будут находиться в нем постоянно. Нам нужен механизм, обеспечивающий обновление кэшируемых результатов (периодически или «по запросу») при изменении определенного контента.

Чтобы ответить на этот вопрос, необходимо сначала разобраться, когда происходит изменение результатов (обсудите этот вопрос с интервьюером). Основные случаи:

1. Изменяется контент страницы, расположенной по заданному адресу (или страница удалена).
2. Результаты упорядочиваются на основании ранга страницы (который может изменяться).
3. Появляются новые страницы, связанные с запросом.

Чтобы обработать случаи 1 и 2, можно создать отдельную хеш-таблицу, которая покажет, какие кэшированные запросы связаны с конкретным URL-адресом. Данную задачу можно решать отдельно от других кэшей, а ресурсы могут размещаться на других компьютерах. С другой стороны, такой подход требует большого количества данных.

Альтернативное решение: если данные не требуют мгновенного обновления (в большинстве случаев это так), можно периодически просматривать кэш, сохраненный на каждой машине, и чистить запросы, связанные с обновлением URL-адресов.

Ситуация 3 создает гораздо больше проблем. Можно обновлять запросы из одного слова, разбирая содержимое нового URL и удаляя однословные запросы из кэшей. Но это решение подходит только для однословных запросов.

Хороший способ обработать ситуацию 3 — предусмотреть «автоматический таймаут» для кэша. Независимо от того, насколько запрос популярен, он *не* должен находиться в кэше больше `x` минут. Тем самым гарантируется периодическое обновление всех данных.

Шаг 4. Дальнейшие усовершенствования

Существует множество улучшений и тонких настроек, реализуемых в проекте и зависящих от ваших предположений и ситуаций, для которых вы производите оптимизацию.

Одна из таких оптимизаций — улучшенная обработка самых популярных запросов. Возьмем радикальный пример: допустим, конкретная строка встречается в 1% всех

запросов. Вместо многократных передач запроса машиной i машине j машина i может передать машине j запрос только один раз, а затем сохранить полученные результаты в собственном кэше.

Можно использовать и другой подход — слегка изменить архитектуру системы, чтобы запросы назначались машинам на основании хеш-кода (а следовательно, местонахождения кэша), а не произвольно. Впрочем, у такого решения есть как достоинства, так и недостатки.

Еще одна оптимизация, которую мы можем реализовать, — автоматический тайм-аут. Как уже говорилось, этот механизм уничтожает любые данные через X минут. Однако некоторые данные (например, новости) нужно обновлять чаще, чем другие (например, архивы курсов акций). Можно реализовать тайм-аут на основании темы или на основании URL-адреса. Во втором случае величина тайм-аута для каждого URL-адреса определяется в зависимости от того, как часто страница обновлялась в прошлом. Тайм-аут запроса равен минимальному тайм-ауту для каждого URL-адреса.

Это лишь некоторые из возможных оптимизаций. Помните, что в таких задачах не существует единственно правильного решения. Такие задания в основном сводятся к обсуждению критериев проектирования с интервьюером и демонстрации используемых подходов и методов.

9.6. Крупный интернет-магазин хочет составить список самых популярных продуктов — в целом и по отдельным категориям. Например, один продукт может стоять на 1056-м месте в общем списке популярности, но при этом занимать 13-е место в категории «Спортивное оборудование» и 24-е место в категории «Безопасность». Опишите, как бы вы подошли к проектированию такой системы.

РЕШЕНИЕ

Сначала сделаем некоторые предположения для уточнения формулировки задачи.

Шаг 1. Определение масштаба задачи

Сначала необходимо определить, что же именно мы строим.

- Будем считать, что нам предложено только спроектировать компоненты, относящиеся к задаче, а не полноценную систему электронной коммерации. В данном случае мы затронем вопросы проектирования внешней подсистемы и компонентов оформления покупки, но лишь в той степени, в какой они влияют на рейтинги продаж.
- Также следует определить, что подразумевается под «рейтингом продаж». Общие продажи за все время? Продажи за последний месяц? Последнюю неделю? Какая-нибудь более сложная функция (например, с экспоненциальным затуханием значимости данных продаж)? Эту тему следует обсудить с интервьюером. Будем считать, что в нашем примере используется объем продаж за прошлую неделю.
- Будем считать, что каждый продукт может принадлежать нескольким категориям, а концепции «подкатегорий» не существует.

Эта часть дает хорошее представление о задаче и о масштабе требуемой функциональности.

Шаг 2. Разумные предположения

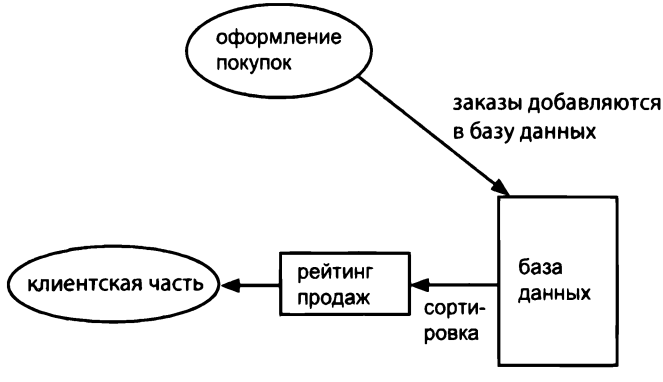
В этом разделе представлены аспекты, которые стоит обсудить с интервьюером. Сейчас перед нами интервьюера нет, так что придется руководствоваться некоторыми предположениями.

- Будем считать, что статистика не обязана быть актуальной на 100%. Данные по самым популярным товарам (например, первая сотня в каждой категории) должны быть получены за последний час, а для менее популярных товаров — за последние сутки. Другими словами, мало кого волнует, если 2 809 132-й по популярности товар на самом деле должен быть 2 789 158-м.
- Точность важна для самых популярных товаров, но для менее популярных позиций допустима небольшая погрешность.
- Предполагается, что данные должны обновляться каждый час (для самых популярных товаров), но временной диапазон таких данных не обязан составлять ровно последние семь дней (168 часов). Если диапазон в отдельных случаях сокращается до 150 часов, это нормально.
- Будем считать, что разбивка на категории основана исключительно на происхождении операции (то есть названии компании-продавца), а не на цене и дате.

Здесь важны не столько конкретные решения, которые вы примете по каждому отдельному вопросу, а ваше понимание того, что это именно предположения. Постарайтесь с самого начала прояснить как можно больше предположений. Может оказаться, что в процессе работы вам придется делать дополнительные предположения.

Шаг 3. Определение основных компонентов

Теперь следует спроектировать простую примитивную схему с описанием основных компонентов. Пора взять маркер и подойти к доске.



В этой простой схеме каждый заказ сохраняется сразу же после его поступления в базу данных. Каждый час или около того мы извлекаем данные из базы по

категориям, вычисляем общие данные продаж, сортируем их и сохраняем в некоем кэше для данных рейтингов продаж (вероятно, находящемся в памяти). Клиентская часть просто извлекает рейтинг продаж из таблицы — вместо того, чтобы обращаться к стандартной базе данных и проводить аналитику самостоятельно.

Шаг 4: Выявление основных проблем

Аналитика обходится дорого

В нашей примитивной системе из базы данных периодически запрашиваются данные по объемам продаж за последнюю неделю по каждому продукту. Такие запросы могут быть достаточно затратными: вы выполняете запрос по всем продажам за все время.

В базе данных достаточно отслеживать только общие объемы продаж. Будем считать (как упомянуто в начале решения), что общее хранение истории покупок реализовано в других частях системы, а мы сосредоточимся лишь на аналитике данных продаж.

Вместо того чтобы производить выборку всех покупок, мы сохраним в базе данных общие объемы продаж за последнюю неделю. Каждая покупка приводит к обновлению суммарных данных продаж за неделю.

Над отслеживанием общих продаж также необходимо подумать. Если мы просто используем специальный столбец для отслеживания общих продаж за последнюю неделю, то данные придется ежедневно пересчитывать (потому что изменяются дни, входящие в последнюю неделю). Пересчет потребует неоправданно высоких затрат.

Вместо этого лучше воспользоваться таблицей следующего вида:

Идентификатор продукта	Сумма	Пн	Вт	Ср	Чт	Пт	Сб	Вс

Фактически таблица работает по принципу циклического массива: каждый день очищается соответствующий день недели. При покупке обновляется как объем продаж продукта за этот день, так и общая сумма.

Также потребуется отдельная таблица для хранения связей идентификаторов продукта с идентификаторами категорий.

Идентификатор продукта	Идентификатор категории

Чтобы получить рейтинг продаж по категории, следует выполнить объединение этих двух таблиц.

Запись в базу данных происходит очень часто

Даже с таким изменением мы будем очень часто обращаться к базе данных. Вероятно, при ежесекундных покупках операции записи в базу данных следует объединять в пакеты.

Вместо того чтобы немедленно закреплять каждую покупку в базе данных, можно сохранять покупки в кэше, находящемся в памяти (а также регистрировать их в журнальном файле на случай сбоев). Система периодически обрабатывает информацию в журнале/кэше, вычисляет суммы и обновляет базу данных.

Также следует прикинуть, насколько возможно хранить данные в памяти. Если в системе 10 миллионов продуктов, возможно ли сохранить каждый из них (вместе со счетчиком) в хеш-таблице? Да. Если идентификатор каждого продукта состоит из 4 байт (что позволяет иметь до 4 миллиардов уникальных идентификаторов), а каждый счетчик тоже состоит из 4 байт (более чем достаточно), такая хеш-таблица займет всего около 40 Мбайт. Даже с дополнительными затратами и при существенном росте системы мы все равно сможем разместить все данные в памяти.

После обновления базы данных можно заново обработать данные рейтингов продаж. Впрочем, здесь необходима осторожность. Если статистика будет вычисляться между обработкой данных двух продуктов, в данных может возникнуть смещение (так как для одного продукта учитывается больший промежуток времени, чем для конкурирующего» продукта).

Как решить эту проблему? Либо проследите за тем, чтобы рейтинги продаж вычислялись только после обработки всех хранимых данных (хотя это нелегко сделать при постоянном поступлении новых покупок), либо организуйте деление кэша в памяти по периодам времени. Если в определенный момент времени база данных будет обновляться для всей хранимой информации, это гарантирует отсутствие искажений.

Соединения обходятся дорого

Теоретически в системе могут существовать десятки тысяч категорий продуктов. Для каждой категории необходимо сначала извлечь данные для входящих в нее товаров (вероятно, с использованием затратной операции соединения), а потом отсортировать их.

Также можно выполнить всего одно соединение продуктов и категорий, чтобы каждый продукт входил один раз в каждую из своих категорий. Если после этого провести сортировку по категории и идентификатору продукта, мы сможем перебрать результаты и получить рейтинги продаж по каждой категории.

Идентификатор продукта	Категория	Сумма	Пн	Вт	Ср	Чт	Пт	Сб	Вс
1423	sportseq	13	4	1	4	19	322	32	232
1423	safety	13	4	1	4	19	322	32	232

Вместо того чтобы выполнять тысячи запросов (по одному для каждой категории), можно отсортировать данные сначала по категории, а потом по объему продаж. Затем при обработке полученных результатов можно получить рейтинги продаж

по каждой категории. Также необходимо провести сортировку всей таблицы по объему продаж для получения общего рейтинга.

Наконец, данные можно с самого начала хранить в таблице, имеющей такую структуру, — вместо того, чтобы выполнять соединения. В этом случае для каждого продукта придется обновлять сразу несколько строк.

Запросы к базам данных все равно могут обходиться дорого

Если запросы и операции записи обходятся слишком дорого, можно полностью отказаться от базы данных и записывать данные в журнальные файлы. В этом случае будут использоваться преимущества такой технологии, как MapReduce.

В такой системе информация о покупке записывается в простой текстовый файл с идентификатором продукта и временной меткой. Для каждой категории создается отдельный каталог, а каждая покупка записывается во все категории, связанные с этим продуктом.

В системе часто выполняются задания для слияния файлов по идентификатору продукта и временным диапазонам, чтобы все покупки для заданного дня (а возможно, и часа) были сгруппированы.

```
/sportsequipment
  1423,Dec 13 08:23-Dec 13 08:23,1
  4221,Dec 13 15:22-Dec 15 15:45,5
  ...
/safety
  1423,Dec 13 08:23-Dec 13 08:23,1
  5221,Dec 12 03:19-Dec 12 03:28,19
...
```

Чтобы получить список ведущих продуктов по каждой категории, достаточно отсортировать каждый каталог.

Как получить общий рейтинг? Есть два хороших решения:

- Можно рассматривать «общую» категорию как очередной каталог и записывать в этот каталог данные каждой покупки. Конечно, это будет означать, что в этом каталоге будет храниться огромное количество файлов.
- Или, поскольку в каждой категории продукты уже отсортированы по объему продаж, также можно провести N-стороннее слияние для получения общего рейтинга.

Также можно воспользоваться тем фактом, что данные не обязаны быть на 100% актуальными (как упоминалось ранее). Актуальными должны быть лишь самые популярные позиции.

Самые популярные товары из каждой категории (первые 100 или около того) можно объединить попарно. После получения 100 элементов в этом порядке сортировки можно прервать слияние для этой пары и перейти к следующей.

Для получения рейтинга по всем продуктам эту работу можно проводить намного реже — например, раз в день.

Среди преимуществ такого решения можно отметить эффективное масштабирование. Так как файлы не зависят друг от друга, их можно легко распределить по нескольким серверам.

Дальнейшие вопросы

Интервьюер может направить процесс проектирования по разным путям.

- Как вы думаете, где в вашей системе возникнут «узкие места»? Что с ними делать?
- А если бы в системе поддерживались подкатегории (чтобы товары могли входить как в категорию «Спорт», так и в категорию «Спортивное оборудование» или даже «Спорт ▸ Спортивное оборудование ▸ Теннис ▸ Ракетки»)?)
- А если данные должны быть более точными? Что, если потребуется обеспечить 30-минутную точность для всех продуктов?

Внимательно обдумайте свой проект, проанализируйте его сильные и слабые стороны. Возможно, вам предложат более подробно проработать какие-то отдельные аспекты.

9.7. Объясните, как бы вы спроектировали сервис персонального финансового менеджера (аналог Mint.com). Система должна подключаться к банковским счетам, анализировать характер расходов и давать рекомендации.

РЕШЕНИЕ

Прежде всего необходимо определить, что же именно мы строим.

Шаг 1. Определение масштаба задачи

Как обычно, уточните требования к системе у интервьюера. Допустим, масштаб задачи был определен следующим образом:

- Вы создаете учетную запись и добавляете к ней банковские счета (которых может быть несколько). Также следует предусмотреть возможность добавления банковских счетов в будущем.
- Система загружает всю вашу финансовую историю (или столько, сколько позволит банк).
- В финансовую историю включаются расходы (купленные или оплаченные товары), доходы (зарплата и другие поступления) и текущие денежные средства (находящиеся на вашем банковском счете или заложенные в инвестиции).
- Каждой операции назначается некоторая «категория» (еда, путешествия, одежда и т. д.).
- Существует источник данных, который сообщает системе (с некоторой надежностью), с какой категорией связана операция. В некоторых случаях пользователь может переопределить неправильно назначенную категорию (скажем, если посещение кафе в торговом центре отнесено к категории «одежда» вместо категории «еда»).

- Система может выдавать пользователю рекомендации относительно его расходов. Рекомендации формируются на основании подборки «типичных» пользователей («люди обычно не тратят более X% своего дохода на одежду»), но могут переопределяться на уровне конкретных бюджетов. На данный момент этот аспект для нас не столь важен.
- Будем считать, что система реализована в виде веб-сайта, хотя теоретически также можно рассмотреть возможность реализации в виде мобильного приложения.
- Система должна отправлять уведомления пользователю по электронной почте — периодически или при выполнении определенных условий (достижение заданного порога расходов и т. д.).
- Будем считать, что в системе не предусмотрена возможность определения пользовательских правил назначения категорий операциям.

Эти требования приблизительно описывают то, что мы хотим построить.

Шаг 2. Разумные предположения

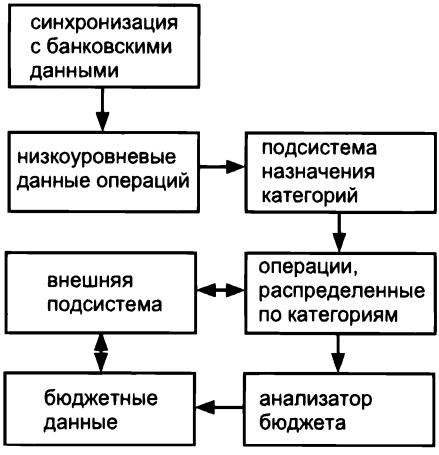
Разобравшись с основными целями для построения системы, следует принять дальнейшие предположения относительно ее характеристик.

- Операции добавления и удаления банковских счетов выполняются относительно редко.
- В системе интенсивно выполняются операции записи. Типичный пользователь может совершать несколько операций ежедневно, хотя лишь немногие пользователи будут обращаться к сайту чаще раза в неделю. Предполагается, что для многих пользователей основные взаимодействия будет сводиться к уведомлениям по электронной почте.
- После того как операции будет назначена категория, она может быть изменена только в том случае, если пользователь явно потребует изменить ее. Система никогда не переназначает категории автоматически, даже если правила изменятся. Это означает, что двум идентичным операциям могут быть назначены разные категории, если правила изменятся между операциями. Такое решение объясняется тем, что автоматическое изменение категории без участия пользователя может сбить пользователя с толку.
- Скорее всего, банки не будут активно поставлять данные в нашу систему. Вместо этого данные придется запрашивать у них.
- Уведомления о превышении бюджета пользователем не обязательно отправлять мгновенно (да это и нереально, потому что информация об операциях не будет поступать мгновенно). Вероятно, максимальная задержка до 24 часов не создаст проблем.

Ничто не мешает вам сделать другие предположения, но вы должны явно изложить их своему интервьюеру.

Шаг 3. Определение основных компонентов

Самая примитивная реализация извлекает банковские данные при каждом входе, классифицирует данные и анализирует бюджет пользователя. Однако такое решение не в полной мере соответствует требованиям, так как мы хотим организовать уведомления о событиях по электронное почте. Конечно, можно добиться большего.



При такой базовой архитектуре банковские данные загружаются с определенной периодичностью (ежечасно или ежедневно). Частота может зависеть от поведения пользователей. У менее активных пользователей проверка счетов может выполняться реже.

После появления новые данные сохраняются в списке необработанных операций. Затем эти данные передаются подсистеме назначения категорий, которая назначает каждой операции категорию и сохраняет классифицированные операции в другом хранилище.

Анализатор бюджета читает операции, распределенные по категориям, обновляет бюджет пользователя по категориям и сохраняет данные бюджета.

Внешняя подсистема запрашивает данные как из хранилища операций, распределенных по категориям, так и из хранилища бюджетных данных. Кроме того, пользователь может взаимодействовать с внешней подсистемой, изменяя бюджет или распределение своих операций по категориям.

Шаг 4. Выявление основных проблем

Теперь следует проанализировать основные проблемы, которые могут возникнуть в этой архитектуре.

Система обрабатывает большое количество данных. С другой стороны, мы хотим, чтобы она была быстрой и мгновенно реагировала на действия пользователя, поэтому как можно большая часть обработки должна производиться асинхронно.

Почти наверняка должна существовать как минимум одна очередь для таких задач, как загрузка новых банковских данных, повторный анализ бюджетов и распределение новых банковских данных по категориям. Также в очередь

должны повторно ставиться задачи, предыдущие попытки выполнения которых завершились неудачей.

Вероятно, задачам должны назначаться приоритеты, так как некоторые из них должны выполняться чаще других. Желательно построить систему очередей, в которых одни задачи будут планироваться для ускоренного выполнения, но при этом все задачи рано или поздно будут выполнены. Другими словами, низкоприоритетные задачи не должны бесконечно простаивать, потому что в любой момент находится задача с более высоким приоритетом.

Одним из важных компонентов системы, которые нами еще не рассматривались, является подсистема электронной почты. Теоретически можно создать задачу для регулярной обработки пользовательских данных и проверки превышения бюджета, но это будет означать ежедневную проверку каждого пользователя. Вместо этого следует ставить задачу в очередь при выполнении операций, способных привести к превышению бюджета. Можно организовать хранение текущих расходов по категориям, чтобы было проще понять, приведет ли новая операция к превышению бюджета.

Также следует рассмотреть тот факт (или предположение), что в подобных системах часто присутствует большое количество неактивных пользователей, то есть пользователей, которые ни разу не обращаются к системе с момента своей регистрации. Таких пользователей следует либо полностью удалять из системы, либо снижать приоритеты их учетных записей. Значит, понадобится механизм отслеживания активности учетных записей и назначения им приоритетов.

Скорее всего, главным «узким местом» будет огромный объем данных, которые система должна загрузить и проанализировать. Следует предусмотреть возможность асинхронного получения банковских данных и распределения этих задач по многим серверам. Подсистему разделения по категориям и бюджетный анализатор следует рассмотреть чуть подробнее.

Подсистема распределения по категориям и бюджетный анализатор

Важно, что операции не зависят друг от друга. Получив операцию пользователя, мы можем назначить ей категорию и интегрировать ее данные. Возможно, это не самый эффективный подход, но он не угрожает целостности данных.

Нужно ли использовать при этом стандартную базу данных? При большом количестве одновременно поступающих операций это может быть неэффективно. Большое количество соединений явно нежелательно.

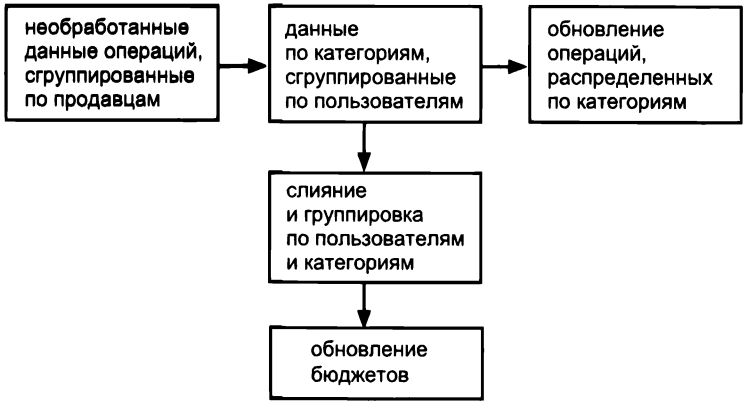
Возможно, будет лучше просто сохранять операции в наборе текстовых файлов. Ранее предполагалось, что категории назначаются исключительно по названию продавца. При большом количестве пользователей между продавцами будет появляться большое количество дубликатов. Группировка файлов операций по названию продавца позволит воспользоваться этими дубликатами.

Подсистема назначения категорий может работать примерно так (см. схему ниже).

Сначала подсистема получает необработанные данные операций, сгруппированные по продавцам. Она выбирает категорию, соответствующую продавцу (которая

может храниться в кэше для часто используемых продавцов), и назначает ее этим операциям.

После применения категории все операции пользователя группируются заново по пользователям, после чего каждая операция вставляется в базу данных для соответствующего пользователя.



До назначения категорий	После назначения категорий
amazon/ user121,$5.43,Aug 13 user922,$15.39,Aug 27 ... comcast/ user922,$9.29,Aug 24 user248,$40.13,Aug 18 ...	user121/ amazon,shopping,$5.43,Aug 13 ... user922/ amazon,shopping,$15.39,Aug 27 comcast,utilities,$9.29,Aug 24 ... user248/ comcast,utilities,$40.13,Aug 18 ...

Затем в игру вступает анализатор бюджета. Он берет данные, сгруппированные по пользователям, производит их слияние по категориям и обновляет бюджет.

Большая часть этих задач решается на уровне простых журнальных файлов. В базе данных будут сохранены только итоговые данные (операции, распределенные по категориям, и анализ бюджета). Тем самым сводится к минимуму количество операций чтения и записи в базу данных.

Изменение категорий пользователем

Пользователь может избирательно переопределять категории отдельных операций. В этом случае обновляется хранилище данных, а также инициируется быстрый пересчет бюджета для исключения элемента из старой категории и внесения его в другую категорию.

Также можно пересчитать бюджет с нуля. Анализатор бюджета работает довольно быстро, потому что ему достаточно проанализировать операции одного пользователя за несколько недель.

Дальнейшие вопросы

- Как изменится архитектура, если потребуется обеспечить работу мобильного приложения?
- Как бы вы спроектировали компонент, распределяющий элементы между категориями?
- Как бы вы спроектировали функцию бюджетных рекомендаций?
- Как изменится архитектура, если пользователь может переопределять категории всех операций с конкретным продавцом?

9.8. Спроектируйте систему — аналог Pastebin, в которой пользователь может ввести фрагмент текста и получить случайно сгенерированный URL-адрес для обращения к нему.

РЕШЕНИЕ

Начнем с прояснения требований к системе.

Шаг 1. Определение масштаба задачи

- Система не поддерживает учетные записи пользователей или редактирование документов.
- Система отслеживает статистику количества обращений к каждой странице.
- Старые документы удаляются при отсутствии обращений в течение достаточно длительного периода времени.
- Несмотря на отсутствие полноценной аутентификации при обращении к документам, у пользователей не должно быть возможности легко «угадать» URL-адреса документов.
- Система предоставляет как внешний интерфейс, так и API.
- Статистика по каждому URL-адресу доступна по ссылке на странице. По умолчанию эта ссылка не отображается.

Шаг 2. Разумные предположения

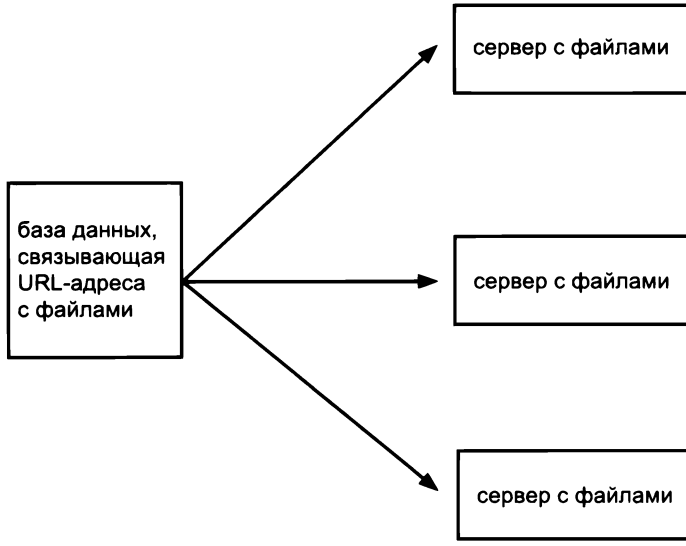
- Система обрабатывает интенсивный трафик и содержит миллионы документов.
- Распределение трафика между документами неравномерно. К некоторым документам пользователи обращаются намного чаще, чем к другим.

Шаг 3. Определение основных компонентов

Попробуем набросать простую схему. Система должна отслеживать URL-адреса и файлы, связанные с ними, а также вести статистику частоты обращений к файлам.

Как организовать хранение документов? Есть два варианта: хранение в базе данных и хранение в файле. Так как документы могут быть большими, а поиск вряд ли понадобится, вероятно, вариант с хранением в файле является предпочтительным.

Простая архитектура на следующей схеме может оказаться вполне эффективной:



Простая база данных определяет местонахождение (сервер и путь) каждого файла. При получении запроса URL-адреса система определяет местонахождение данных по базе, а затем обращается к соответствующему файлу.

Также потребуется база данных для хранения статистики. Для этого подойдет простая база данных, в которой сохраняется информация о каждом посещении (включая временную метку, IP-адрес и местонахождение). Когда системе потребуется использовать статистику каждого посещения, она читает соответствующие данные из базы.

Шаг 4. Выявление основных проблем

Первая проблема, которая приходит в голову, — что к некоторым документам пользователи будут обращаться намного чаще, чем к другим. Чтение данных из файловой системы выполняется относительно медленно (по сравнению с чтением из памяти). Вероятно, в систему стоит включить кэш для хранения документов, использовавшихся в последнее время. Это гарантирует высокую скорость доступа к часто используемым (или использовавшимся в последнее время) документам.

Так как документы не могут редактироваться, нам не нужно беспокоиться о том, что содержимое кэша может стать недействительным.

Также можно рассмотреть потенциальную возможность сегментации (sharding) базы данных. Сегментация может быть построена на основе отображения URL-адреса (например, вычислением остатка от деления хеш-кода URL на некоторое целое число), позволяющего быстро найти базу данных с нужным файлом.

Можно пойти еще дальше: вообще отказаться от базы данных и определять сервер, на котором хранится документ, на основании хеш-кода URL. Один из потенциальных недостатков такой схемы заключается в том, что если в систему будут добавлены новые серверы, могут возникнуть трудности с перераспределением документов.

Генерирование URL

Мы еще не обсудили, как должны генерироваться URL-адреса. Вероятно, использовать монотонно возрастающие целочисленные значения не стоит — пользователю будет слишком легко «подобрать» URL существующего документа. Мы хотим, чтобы без ссылки URL-адрес было достаточно трудно найти.

Одно из простых решений — сгенерировать случайный код GUID (например, 5d50e8ac-57cb-4a0d-8661-bcdee2548979). Уникальность этих 128-разрядных значений формально не гарантирована, но вероятность совпадения настолько мала, что GUID можно считать уникальными. Недостаток такого решения заключается в том, что URL-адрес получается «некрасивым». Можно хешировать его в меньшее значение, но тогда повышается вероятность коллизий.

Также можно воспользоваться очень похожим решением: сгенерировать 10-символьную последовательность из цифр и букв, что дает 36^{10} возможных комбинаций. Даже с милиардом URL вероятность коллизий по URL-адресам чрезвычайно низка.

Это не значит, что вероятность коллизий в масштабах всей системы низка. Это не так: просто коллизия по каждому отдельному URL маловероятна. Однако после сохранения миллиарда URL-адресов вероятность появления коллизии в какой-то момент становится вполне заметной.

Если предположить, что периодическая (пусть и маловероятная) потеря данных нас решительно не устраивает, необходимо как-то обрабатывать коллизии. Можно либо обратиться к хранилищу данных и посмотреть, существует ли в нем сгенерированный URL-адрес, либо, если URL отображается на конкретный сервер, просто проверить, существует ли в заданном месте файл. При обнаружении коллизии можно просто сгенерировать новый URL. С 36^{10} возможных URL-адресов коллизии будут настолько редкими, что даже «ленивого» решения (обнаружение коллизии и повторная попытка) будет достаточно.

Статистика

Осталось рассмотреть последний компонент — блок статистики. Вероятно, для документов следует хранить количество обращений, возможно, с разбивкой по местонахождению или времени.

Здесь также возможны два варианта:

- Хранение полного набора данных о каждом посещении.

- Хранение только тех данных, которые заведомо будут использоваться (количество посещений и т. д.).

Вы можете обсудить эту тему с интервьюером, но скорее всего, хранение полного набора данных более разумно. Никогда не знаешь, какие возможности потребуется реализовать в аналитике в будущем. Полные данные обеспечивают эту гибкость.

Это не означает, что система должна предоставлять простые средства поиска по полным данным (или хотя бы предоставлять доступ к ним). Можно просто регистрировать информацию о каждом посещении в журнале и передавать информацию на другие серверы.

Следует учитывать, что объем таких данных может быть достаточно существенным. Затраты пространства можно значительно сократить за счет вероятностного хранения данных. С каждым URL-адресом связывается метрика вероятности, которая уменьшается с ростом популярности сайта. Например, данные популярного документа могут сохраняться всего один раз из десяти (в среднем). При выборке количества посещений значение должно умножаться на величину, обратную вероятности (например, на 10 в приведенном примере). Конечно, такой способ породит некоторую погрешность, но она может оказаться допустимой.

Файлы журналов не предназначены для частого использования. Также возможно хранение обработанных данных в хранилище. Если в разделе статистики отображается только количество посещений и график в зависимости от времени, информация может храниться в отдельной базе данных.

URL	Месяц и год	Посещения
12ab31b92p	декабрь 2013	242119
12ab31b92p	январь 2014	429918
...	...	...

При каждом обращении к URL-адресу увеличивается значение соответствующей строки и столбца. Также возможна сегментация базы данных по URL-адресам.

Так как статистика не выводится на обычных страницах и не представляет интереса для рядового пользователя, она не должна порождать значительной нагрузки. Впрочем, генерируемая разметка HTML все равно может кэшироваться на серверах внешней подсистемы, чтобы избежать частых обращений к данным для самых популярных URL-адресов.

Дальнейшие вопросы

- Как бы вы реализовали поддержку учетных записей пользователей?
- Как бы вы добавили на страницу статистики новый тип аналитики (например, источник перехода по ссылке)?
- Как бы вы изменили структуру системы, если бы статистика отображалась вместе с каждым документом?

10

Сортировка и поиск

10.1. Имеются два отсортированных массива A и B. В конце массива A существует свободное место, достаточное для размещения массива B. Напишите метод слияния массивов B и A, сохраняющий сортировку.

РЕШЕНИЕ

Так как мы знаем, что в конце массива A имеется буфер достаточного размера, нет необходимости выделять дополнительное пространство. Логика проста: нужно сравнивать элементы A и B и расставлять их, упорядочивая, пока элементы в A и B не закончатся.

Единственная проблема: при вставке элемента в начало массива A необходимо сдвинуть остальные элементы, чтобы освободить для него место. Лучше было бы добавлять элементы в конец массива, где есть свободное место.

Приведенный далее код реализует этот алгоритм. Он начинает работать с последних элементов в массивах A и B и перемещает наибольшие элементы в конец A.

```
1  void merge(int[] a, int[] b, int lastA, int lastB) {
2    int indexA = lastA - 1; /* Индекс последнего элемента a */
3    int indexB = lastB - 1; /* Индекс последнего элемента b */
4    int indexMerged = lastB + lastA - 1; /* Конец объединенного массива */
5
6    /* Слияние a и b, начиная с последних элементов */
7    while (indexB >= 0) {
8      /* конец a > конец b */
9      if (indexA >= 0 && a[indexA] > b[indexB]) {
10       a[indexMerged] = a[indexA]; // Копирование элемента
11       indexA--;
12     } else {
13       a[indexMerged] = b[indexB]; // Копирование элемента
14       indexB--;
15     }
16     indexMerged--; // Перемещение индексов
17   }
18 }
```

Обратите внимание: после того как все элементы B будут расставлены, вам не придется копировать содержимое A. Все элементы окажутся на своих местах.

10.2. Напишите метод сортировки массива строк, при котором анаграммы группируются друг за другом.

РЕШЕНИЕ

В этой задаче нужно сгруппировать строки в массиве так, чтобы анаграммы следовали одна за другой. Обратите внимание: никакой другой конкретный порядок слов не требуется.

Как известно, анаграммами называются слова, состоящие из одинаковых символов, но расположенных в разном порядке. Следовательно, если расположить символы в одинаковом порядке, можно легко определить, являются ли новые слова одинаковыми.

Одно из возможных решений задачи — применить любой стандартный алгоритм сортировки (например, сортировку слиянием или быструю сортировку), но изменить алгоритм сравнения (компаратор). Компаратор позволяет проверить эквивалентность двух строк, являющихся потенциальными анаграммами.

Как проще всего проверить, являются ли слова анаграммами? Можно подсчитать частоту использования разных символов в каждой строке и возвратить `true`, если эти значения совпадают. С другой стороны, строки можно просто отсортировать; если после сортировки будут получены два одинаковых результата, значит, строки являются анаграммами.

В следующем коде реализуется интерфейс компаратора:

```
1  public class AnagramComparator implements Comparator<String> {
2      public String sortChars(String s) {
3          char[] content = s.toCharArray();
4          Arrays.sort(content);
5          return new String(content);
6      }
7
8      public int compare(String s1, String s2) {
9      return sortChars(s1).compareTo(sortChars(s2));
10     }
11 }
```

Теперь достаточно отсортировать массив, используя метод `compareTo` вместо обычного:

```
12 Arrays.sort(array, new AnagramComparator());
```

Алгоритм занимает $O(n \log(n))$ времени.

Это лучший возможный результат, который дает общий алгоритм сортировки, но на самом деле нам сортировать массив полностью не обязательно. Нужно *сгруппировать* строки в массиве в соответствии с анаграммами.

Для этого можно воспользоваться хеш-таблицей, которая отображает отсортированную версию слова на список его анаграмм. Так, например, `acre` будет соответствовать списку слов {`acre`, `race`, `care`}. Как только все слова в таких списках будут сгруппированы, их можно будет поместить обратно в массив.

Следующий код реализует этот алгоритм:

```
1  void sort(String[] array) {
2    HashMapList<String, String> mapList = new HashMapList<String, String>();
3
4    /* Группировка слов по анаграммам */
```

```
5    for (String s : array) {
6      String key = sortChars(s);
7      mapList.put(key, s);
8    }
9
10    /* Преобразование хеш-таблицы в массив */
11    int index = 0;
12    for (String key : mapList.keySet()) {
13      ArrayList<String> list = mapList.get(key);
14      for (String t : list) {
15        array[index] = t;
16        index++;
17     }
18    }
19  }
20
21  String sortChars(String s) {
22    char[] content = s.toCharArray();
23    Arrays.sort(content);
24    return new String(content);
25  }
26
27  /* HashMapList<String, Integer> связывает String с
28   * ArrayList<Integer>. Реализация приведена в приложении. */
```

Заметим, что этот алгоритм представляет собой разновидность алгоритма блочной сортировки.

10.3. Имеется отсортированный массив из n целых чисел, который был циклически сдвинут неизвестное число раз. Напишите код для поиска элемента в массиве. Предполагается, что массив изначально был отсортирован по возрастанию.

Пример:

Ввод: найти 5 в {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Вывод: 8 (индекс числа 5 в массиве)

РЕШЕНИЕ

Если у вас возникла мысль о бинарном поиске, вы правы!

В классической задаче бинарного поиска `x` сравнивается с центральной точкой, чтобы узнать, где находится `x` — слева или справа. Сложность в нашем случае заключается в том, что массив был циклически сдвинут, а значит, может иметь точку перегиба. Рассмотрим, например, два следующих массива:

```
Array1: {10, 15, 20, 0, 5}
Array2: {50, 5, 20, 30, 40}
```

У обоих массивов есть средняя точка — 20, но в первом случае 5 находится справа от нее, а в другом — слева. Поэтому сравнения `x` со средней точкой недостаточно.

Однако если разобраться поглубже, можно заметить, что половина массива упорядочена нормально (то есть по возрастанию). Следовательно, мы можем рассмотреть отсортированную половину массива и решить, где именно следует производить поиск — в левой или правой половине.

Например, если мы ищем 5 в `Array1`, то посмотрим на крайний левый элемент (10) и средний элемент (20). Так как 10 < 20, становится понятно, что левая половина отсортирована. Поскольку 5 не попадает в этот диапазон, мы знаем, что искомое место находится в правой половине массива.

В `Array2` мы видим, что 50 > 20, а значит, отсортирована правая половина. Мы обращаемся к середине (20) и крайнему правому элементу (40), чтобы проверить, попадает ли 5 в рассматриваемый диапазон. Значение 5 не может находиться в правой части, а значит, его нужно искать в левой части массива.

Задача становится более сложной, если левый и средний элементы идентичны, как в случае с массивом `{2, 2, 2, 3, 4, 2}`. В этом случае можно выполнить сравнение с правым элементом и, если он отличается, ограничить поиск правой половиной. В противном случае выбора не остается, и придется анализировать обе половины.

```
int search(int a[], int left, int right, int x) {
  int mid = (left + right) / 2;
  if (x == a[mid]) { // Элемент найден
    return mid;
  }
  if (right < left) {
    return -1;
  }

   /* Либо левая, либо правая половина должна быть нормально упорядочена.
    * Найти нормально упорядоченную сторону, а затем использовать ее
    * для определения стороны, в которой следует искать x. */
   if (a[left] < a[mid]) { // Левая половина нормально упорядочена.
     if (x >= a[left] && x < a[mid]) {
       return search(a, left, mid - 1, x); // Искать слева
     } else {
       return search(a, mid + 1, right, x); // Искать справа
     }
   } else if (a[mid] < a[left]) { // Правая половина нормально упорядочена.
     if (x > a[mid] && x <= a[right]) {
       return search(a, mid + 1, right, x); // Искать справа
     } else {
       return search(a, left, mid - 1, x); // Искать слева
     }
   } else if (a[left] == a[mid]) { // Левая половина состоит из повторов
     if (a[mid] != a[right]) { // Если правая половина отличается, искать в ней
       return search(a, mid + 1, right, x); // Искать справа
     } else { // Иначе искать придется в обеих половинах
       int result = search(a, left, mid - 1, x); // Искать слева
       if (result == -1) {
         return search(a, mid + 1, right, x); // Искать справа
       } else {
         return result;
       }
     }
   }
   return -1;
}
```

Код выполняется за время $O(\log n)$, если все элементы будут разными. При большом количестве дубликатов алгоритм потребует времени $O(n)$. Это объясняется тем, что при большом количестве дубликатов часто приходится обрабатывать обе половины массива (левую и правую).

Концептуально данная задача не сложна, но написать идеальную реализацию достаточно трудно. Не беспокойтесь, если вы допустите пару-тройку ошибок при решении этой задачи. Из-за высокого риска смещения на 1 и других второстепенных ошибок код необходимо тщательно протестировать.

10.4. Имеется структура данных `Listy` — аналог массива, в котором отсутствует метод определения размера. При этом поддерживается метод `elementAt(i)`, возвращающий элемент с индексом *i* за время $O(1)$. Если значение `i` выходит за границу структуры данных, метод возвращает –1. (По этой причине в структуре данных могут храниться только положительные целые числа.) Для заданного экземпляра `Listy`, содержащего отсортированные положительные числа, найдите индекс элемента с заданным значением `x`. Если `x` входит в структуру данных многократно, можно вернуть любой индекс.

РЕШЕНИЕ

Первое, что приходит в голову, — воспользоваться бинарным поиском. Однако для бинарного поиска необходимо знать длину списка, чтобы вычислить его середину. В условиях задачи метод для получения длины отсутствует. Можно ли вычислить длину? Да!

Известно, что `elementAt` возвращает –1 для слишком большого значения. Следовательно, мы можем проверять увеличивающиеся значения, пока не выйдем за границу списка.

Но насколько увеличивающиеся? Если идти по списку линейно — 1, потом 2, потом 3 и т. д., — алгоритм будет работать с линейным временем. Вероятно, нужно реализовать что-то более эффективное. Иначе зачем бы интервьюер упомянул о том, что список отсортирован?

Лучше увеличивать значения по экспоненте: проверить сначала 1, потом 2, потом 4, 8, 16 и т. д. Тем самым гарантируется, что для списка длины `n` длина будет найдена за время не более $O(\log n)$.

Почему именно $O(\log n)$? Представьте, что мы начинаем с указателем `q` в позиции `q=1`. При каждой итерации указатель `q` удваивается, пока не превысит длину `n`. Сколько раз `q` может удвоиться, пока не станет больше `n`? Другими словами, для какого значения `k` выполняется условие $2^k=n$? Условие выполняется при `k=log n` по определению логарифма. Следовательно, для определения длины понадобится $O(\log n)$ шагов.

После того как длина будет определена, выполняется (в основном) обычный бинарный поиск. Я говорю «в основном», потому что необходимо внести одно незначительное изменение: если серединой является –1, это значение следует интерпретировать как «слишком большое» и вести поиск слева (строка 16 в приведенном ниже листинге).

И еще одно небольшое изменение: вспомните, что при определении длины мы вызываем `elementAt` и сравниваем с –1. Если в процессе элемент превысит значение `x` (искомое), мы переходим к бинарному поиску на более ранней стадии.

```
int search(Listy list, int value) {
  int index = 1;
  while (list.elementAt(index) != -1 && list.elementAt(index) < value) {
    index *= 2;
  }
  return binarySearch(list, value, index / 2, index);
}

int binarySearch(Listy list, int value, int low, int high) {
   int mid;

   while (low <= high) {
     mid = (low + high) / 2;
     int middle = list.elementAt(mid);
     if (middle > value || middle == -1) {
       high = mid - 1;
     } else if (middle < value) {
       low = mid + 1;
     } else {
       return mid;
    }
   }
   return -1;
}
```

Выясняется, что отсутствие информации о длине не влияет на время выполнения алгоритма поиска. Длина определяется за время $O(\log n)$, после чего поиск проводится за время $O(\log n)$. Общая временная сложность составляет $O(\log n)$, как и для обычного массива.

10.5. Имеется отсортированный массив строк, в котором могут присутствовать пустые строки. Напишите метод для определения позиции заданной строки.

Пример:

Ввод: строка `"ball"` в массиве `{"at", "", "", "", "ball", "", "","car", "", "", "dad", "", ""}`

Вывод: 4

РЕШЕНИЕ

Если бы не пустые строки, можно было бы просто применить бинарный поиск: сравнить искомую строку (`str`) со средним элементом массива и т. д.

Пустые строки-разделители заставляют реализовать модифицированную версию бинарного поиска. Необходимо исправить сравнение с `mid`, если `mid` оказывается пустой строкой. Для этого достаточно переместить `mid` на ближайшую непустую строку.

Рекурсивный код для решения этой задачи легко преобразуется в итеративный. Такая реализация включена в архив примеров.

```
int search(String[] strings, String str, int first, int last) {
  if (first > last) return -1;
  /* mid устанавливается в середину */
  int mid = (last + first) / 2;

  /* Если mid - пустая строка, найти ближайшую непустую. */
  if (strings[mid].isEmpty()) {
    int left = mid - 1;
    int right = mid + 1;
     while (true) {
       if (left < first && right > last) {
         return -1;
       } else if (right <= last && !strings[right].isEmpty()) {
         mid = right;
         break;
       } else if (left >= first && !strings[left].isEmpty()) {
         mid = left;
         break;
     }
       right++;
       left--;
    }
  }

  /* Проверить строку и выполнить рекурсию при необходимости */
  if (str.equals(strings[mid])) { // Строка найдена!
    return mid;
  } else if (strings[mid].compareTo(str) < 0) { // Искать справа
    return search(strings, str, mid + 1, last);
  } else { // Искать слева
    return search(strings, str, first, mid - 1);
  }
}

int search(String[] strings, String str) {
  if (strings == null || str == null || str == "") {
    return -1;
  }
  return search(strings, str, 0, strings.length - 1);
}
```

Время выполнения этого алгоритма в худшем случае составляет $O(n)$. Более того, эффективность алгоритма в худшем случае не может быть лучше $O(n)$: представьте массив из пустых строк с одной непустой строкой. Не существует «умного» способа найти непустую строку; в худшем случае неизбежно придется проверять каждый элемент массива.

Будьте внимательны с поиском пустых строк. Как следует поступить, нужно ли вам находить расположение пустой строки? (Эта операция потребует времени $O(n)$.) Или следует обработать такую ситуацию как ошибку?

Не существует единственно правильного ответа на эти вопросы. Обсудите эту проблему с интервьюером. Одно то, что вы задали этот вопрос, покажет вашу предусмотрительность.

10.6. Имеется файл размером 20 Гбайт, состоящий из строк. Как бы вы выполнили сортировку такого файла?

РЕШЕНИЕ

Интервьюер наверняка не просто так упомянул о размере 20 Гбайт. Скорее всего, это означает, что данные невозможно загрузить в память одновременно.

Что же делать? Придется загружать в память только часть данных.

Разделим файл на блоки по `X` Мбайт, где `X` — размер доступной памяти. Каждый блок сортируется по отдельности и сохраняется в файловой системе.

Как только все блоки будут отсортированы, можно приступать к слиянию. В результате вы получите полностью отсортированный файл.

Данный алгоритм получил название «внешняя сортировка».

10.7. Имеется входной файл с четырьмя миллиардами неотрицательных целых чисел. Предложите алгоритм для генерирования целого числа, отсутствующего в файле. Считайте, что для выполнения операции доступен 1 Гбайт памяти.

Дополнительно:

А если доступно всего 10 Мбайт памяти? Предполагается, что все значения различны, а количество неотрицательных целых чисел в этом случае не превышает миллиарда.

РЕШЕНИЕ

Всего существуют 2^{32} (или 4 миллиарда) целых чисел и 2^{31} неотрицательных целых чисел. Следовательно, во входном файле (если предположить, что он содержит `int`, а не `long`) присутствуют дубликаты.

Также доступен 1 Гбайт памяти, то есть 8 млрд бит. С 8 млрд бит каждому возможному целому числу можно поставить в соответствие бит в памяти. Логика выглядит так:

1. Создать битовый вектор с 4 миллиардами бит. Битовый вектор — это массив, хранящий в компактном виде булевы переменные (может использоваться как `int`, так и другой тип данных). Каждая переменная типа `int` представляет 32 булевых значения.
2. Инициализировать битовый вектор нулями.
3. Просканировать все числа (`num`) из файла и вызвать `BV.set(num, 1)`.
4. Еще раз просканировать битовый вектор начиная с индекса 0.
5. Вернуть индекс первого элемента со значением 0.

Следующий код демонстрирует работу алгоритма.

```
1  long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
2  byte[] bitfield = new byte [(int) (numberOfInts / 8)];
```

```
String filename = ...

void findOpenNumber() throws FileNotFoundException {
  Scanner in = new Scanner(new FileReader(filename));
  while (in.hasNextInt()) {
    int n = in.nextInt ();
    /* Найти соответствующее число в битовом поле, используя оператор OR
     * для установки n-го бита в байте (например, 10 соответствует
     * 2-му биту индекса 2 в массиве байтов). */
    bitfield [n / 8] |= 1 << (n % 8);
  }

  for (int i = 0; i < bitfield.length; i++) {
    for (int j = 0; j < 8; j++) {
      /* Получение отдельных битов каждого байта. При обнаружении 0 бита
       * вывести соответствующее значение. */
      if ((bitfield[i] & (1 << j)) == 0) {
        System.out.println (i * 8 + j);
        return;
      }
    }
  }
}
```

Дополнительно: а если доступно всего 10 Мбайт?

Отсутствующее число можно найти двойным проходом по набору данных. Разделим целые числа на блоки некоторого размера (мы еще обсудим, как правильно выбрать размер). Пока предположим, что мы используем блоки размером 1000 чисел. Так, `блок0` соответствует числам от 0 до 999, `блок1` — 1000–1999 и т. д.

Нам известно, сколько значений *может* находиться в каждом блоке. Теперь мы анализируем файл и подсчитываем, сколько значений находится в указанном диапазоне: 0–999, 1000–1999 и т. д. Если в диапазоне оказалось 998 значений, то «дефектный» интервал найден.

На втором проходе мы будем искать в этом диапазоне отсутствующее число. Можно воспользоваться концепцией битового вектора из первой части задачи. Числа, не входящие в конкретный диапазон, для нас интереса не представляют.

Остается понять, как выбрать размер блока. Определим несколько переменных:

- Пусть `rangeSize` — размер диапазонов каждого блока на первом проходе.
- Пусть `arraySize` — число блоков при первом проходе. Обратите внимание, что `arraySize` = 2^{31}/`rangeSize`, так как существуют 2^{31} неотрицательных целых чисел.

Значение `rangeSize` следует выбрать так, чтобы памяти хватило и на первый (массив), и на второй (битовый вектор) проходы.

Первый проход: массив

Массив на первом проходе может вместить 10 Мбайт, или приблизительно 2^{23} байт, памяти. Поскольку каждый элемент в массиве относится к типу `int`, а переменная типа `int` занимает 4 байта, мы можем хранить примерно 2^{21} элементов. Из этого следует:

$$\texttt{arraySize} = \frac{2^{31}}{\texttt{rangeSize}} \# 2^{21}$$

$$\texttt{rangeSize} \$ \frac{2^{31}}{2^{21}}$$

$$\texttt{rangeSize} \$ 2^{10}$$

Второй проход: битовый вектор

Нам нужно место, достаточное для хранения `rangeSize` бит. Поскольку в память помещается 2^{23} байт, мы сможем поместить 2^{26} бит в памяти. Таким образом:

$$2^{11} <= \texttt{rangeSize} <= 2^{26}$$

Эти условия дают достаточно пространства для маневра, но чем мы ближе к середине, тем меньше памяти будет использоваться в любой момент времени.

Ниже приведена одна из возможных реализаций этого алгоритма:

```
1  int findOpenNumber(String filename) throws FileNotFoundException {
2    int rangeSize = (1 << 20); // 2^20 бит (2^17 байт)
3
4    /* Подсчет количества значений в каждом блоке. */
5    int[] blocks = getCountPerBlock(filename, rangeSize);
6
7    /* Поиск блока с отсутствующим значением. */
8    int blockIndex = findBlockWithMissing(blocks, rangeSize);
9    if (blockIndex < 0) return -1;
10
11    /* Создание битового вектора для элементов в диапазоне. */
12    byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);
13
14    /* Поиск нуля в битовом векторе */
15    int offset = findZero(bitVector);
16    if (offset < 0) return -1;
17
18    /* Вычисление отсутствующего значения. */
19    return blockIndex * rangeSize + offset;
20  }
21
22  /* Подсчет элементов в каждом диапазоне. */
23  int[] getCountPerBlock(String filename, int rangeSize)
24      throws FileNotFoundException {
25    int arraySize = Integer.MAX_VALUE / rangeSize + 1;
26    int[] blocks = new int[arraySize];
27
28    Scanner in = new Scanner (new FileReader(filename));
29    while (in.hasNextInt()) {
30        int value = in.nextInt();
31        blocks[value / rangeSize]++;
32    }
33    in.close();
34    return blocks;
35  }
36
37  /* Поиск блока с малым значением счетчика. */
38  int findBlockWithMissing(int[] blocks, int rangeSize) {
39    for (int i = 0; i < blocks.length; i++) {
40      if (blocks[i] < rangeSize){
```

```
41        return i;
42      }
43    }
44    return -1;
45  }
46
47  /* Создание битового вектора для значений в заданном диапазоне. */
48  byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
49      throws FileNotFoundException {
50    int startRange = blockIndex * rangeSize;
51    int endRange = startRange + rangeSize;
52    byte[] bitVector = new byte[rangeSize/Byte.SIZE];
53
54    Scanner in = new Scanner(new FileReader(filename));
55    while (in.hasNextInt()) {
56      int value = in.nextInt();
57      /* Если число находится в блоке с отсутствующими числами, сохранить */
58      if (startRange <= value && value < endRange) {
59        int offset = value - startRange;
60        int mask = (1 << (offset % Byte.SIZE));
61        bitVector[offset / Byte.SIZE] |= mask;
62      }
63    }
64    in.close();
65    return bitVector;
66  }
67
68  /* Поиск индекса нулевого бита в байте. */
69  int findZero(byte b) {
70    for (int i = 0; i < Byte.SIZE; i++) {
71      int mask = 1 << i;
72      if ((b & mask) == 0) {
73        return i;
74      }
75    }
76    return -1;
77  }
78
79  /* Поиск нуля в битовом векторе и возвращение индекса. */
80  int findZero(byte[] bitVector) {
81    for (int i = 0; i < bitVector.length; i++) {
82      if (bitVector[i] != ~0) { // If not all 1s
83        int bitIndex = findZero(bitVector[i]);
84        return i * Byte.SIZE + bitIndex;
85      }
86    }
87    return -1;
88  }
```

А если интервьюер дополнительно предложит решить задачу с более жесткими ограничениями на использование памяти? В этом случае придется сделать несколько проходов, используя метод, описанный выше. Сначала проверьте, сколько целых чисел будет найдено в каждом блоке из миллиона элементов. На втором проходе

проверяются числа в блоках из тысячи элементов. Наконец, на третьем проходе можно будет использовать битовый вектор.

10.8. Имеется массив с числами от 1 до N, где N не превышает 32 000. Массив может содержать дубликаты, а значение N заранее не известно. Как вывести все дубликаты из массива, если доступно всего 4 Кбайт памяти?

РЕШЕНИЕ

У нас есть 4 Кбайт памяти, это означает, что мы можем работать с $8 \times 4 \times 2^{10}$ бит. Не забывайте, что 32×2^{10} бит > 32 000. Можно создать битовый вектор размером 32 000 бит, где каждый бит соответствует одному целому числу.

Используя такой битовый вектор, мы можем перебрать элементы массива и пометить каждый элемент v, устанавливая бит v в 1. Обнаружив повторяющееся значение, мы выведем его.

```
1  void checkDuplicates(int[] array) {
2    BitSet bs = new BitSet(32000);
3    for (int i = 0; i < array.length; i++) {
4      int num = array[i];
5      int num0 = num - 1; // Набор битов начинается с 0, числа начинаются с 1
6      if (bs.get(num0)) {
7        System.out.println(num);
8      } else {
9        bs.set(num0);
10     }
11   }
12 }
13
14 class BitSet {
15   int[] bitset;
16
17   public BitSet(int size) {
18     bitset = new int[(size >> 5) + 1]; // Разделить на 32
19   }
20
21   boolean get(int pos) {
22     int wordNumber = (pos >> 5); // Разделить на 32
23     int bitNumber = (pos & 0x1F); // Остаток от деления на 32
24     return (bitset[wordNumber] & (1 << bitNumber)) != 0;
25   }
26
27   void set(int pos) {
28     int wordNumber = (pos >> 5); // Разделить на 32
29     int bitNumber = (pos & 0x1F); // Остаток от деления на 32
30     bitset[wordNumber] |= 1 << bitNumber;
31   }
32 }
```

Задача не особенно сложная, поэтому очень важно реализовать ее без ошибок. Именно поэтому мы реализовали собственный класс для хранения большого битового вектора. Если интервьюер разрешит, можно использовать встроенный в Java класс `BitSet`.

10.9. Для заданной матрицы MxN, в которой каждая строка и столбец отсортированы по возрастанию, напишите метод поиска элемента.

РЕШЕНИЕ

Задачу можно решать двумя способами: «наивное» решение использует только одну часть сортировки, а в более эффективном решении задействованы обе части.

Решение 1. Наивное решение

Чтобы найти нужный элемент, можно воспользоваться бинарным поиском по каждой строке. Алгоритм будет выполняться за время $O(M \log(N))$, так как необходимо обработать M строк, а на поиск в каждой тратится время $O(\log(N))$. Задайте уточняющий вопрос интервьюеру, прежде чем переходить к усовершенствованию алгоритма.

Прежде чем приступать к разработке алгоритма, рассмотрим простой пример:

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Допустим, мы ищем элемент 55. Как его найти?

Если посмотреть на первые элементы строки и столбца, можно начать искать расположение искомого элемента. Очевидно, что 55 не может находиться в столбце, который начинается со значения больше 55, так как в начале столбца всегда находится минимальный элемент. Также мы знаем, что 55 не может находиться правее, так как значение первого элемента каждого столбца увеличивается слева направо. Поэтому, если мы обнаружили, что первый элемент столбца больше x, нужно двигаться влево.

Аналогичную логику можно использовать и для строк. Если мы начали со строки, значение первого элемента которой больше x, нужно двигаться вверх.

Аналогичные рассуждения можно использовать и при анализе последних элементов столбцов или строк. Если последний элемент столбца или строки меньше x, то, чтобы найти x, нужно двигаться вниз (для строк) или направо (для столбцов). Это объясняется тем, что последний элемент всегда максимален.

Соберем воедино все эти наблюдения:

- Если первый элемент столбца больше x, то x находится в столбце, расположенном слева.
- Если последний элемент столбца меньше x, то x находится в столбце, расположенном справа.
- Если первый элемент строки больше x, то x находится в строке, расположенной выше.
- Если последний элемент строки меньше x, то x находится в строке, расположенной ниже.

Начинать можно где угодно, но мы начнем с проверки начальных элементов столбцов.

Мы должны начать с правого столбца и двигаться влево. Это означает, что первым элементом для сравнения будет `array[0][c-1]`, где `c` — количество столбцов. Сравнивая первый элемент столбца с `x` (в нашем случае 55), легко понять, что `x` может находиться в столбцах 0, 1 или 2. Остановимся на `[0][2]`.

Данный элемент не может быть последним элементом строки в полной матрице, но он завершает конец строки в подматрице, а подматрица подчиняется тем же условиям. Элемент `[0][2]` имеет значение 40, то есть он меньше, чем наш элемент, а значит, мы знаем, что нам нужно двигаться вниз.

Теперь подматрица принимает следующий вид (серые ячейки исключены):

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Правила поиска будут применяться многократно. Обратите внимание, что реально используются только правила 1 и 4.

Следующий код реализует этот алгоритм:

```
1  boolean findElement(int[][] matrix, int elem) {
2    int row = 0;
3    int col = matrix[0].length - 1;
4    while (row < matrix.length && col >= 0) {
5      if (matrix[row][col] == elem) {
6        return true;
7      } else if (matrix[row][col] > elem) {
8        col--;
9      } else {
10       row++;
11     }
12   }
13   return false;
14 }
```

Также можно воспользоваться решением, которое в большей степени напоминает бинарный поиск. Код получается заметно более сложным, но строится на тех же принципах.

Решение 2. Бинарный поиск

Еще раз обратимся к нашему примеру.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

Мы хотим воспользоваться свойством сортировки, чтобы повысить эффективность алгоритма. Давайте зададимся вопросом: какую информацию о местонахождении элемента можно получить из уникального свойства упорядочения матрицы?

Известно, что все строки и столбцы отсортированы. Это означает, что элемент `[i] j]` больше, чем элементы в строке `i`, находящиеся между столбцами 0 и `j-1`, и элементы в столбце `j` между строками 0 и `i-1`.

Другими словами:

```
a[i][0] <= a[i][1] <= ... <= a[i][j-1] <= a[i][j]
a[0][j] <= a[1][j] <= ... <= a[i-1][j] <= a[i][j]
```

Посмотрите на матрицу: элемент, который находится в темно-серой ячейке, больше, чем другие выделенные элементы.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

Элементы в светло-серых ячейках упорядочены: каждый из них больше как элементов слева, так и элементов, находящихся выше. Таким образом, по свойству транзитивности темно-серый элемент является наибольшим среди всех элементов в квадрате.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

Это означает, что для любого прямоугольника, выделенного в матрице, самый большой элемент будет находиться в правом нижнем углу.

Аналогично можно сделать вывод о том, что верхний левый угол всегда будет наименьшим. Цвета в приведенной ниже схеме отражают информацию об упорядочивании элементов (светло-серый < темно-серый < черный):

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

А теперь вернемся к исходной задаче: допустим, нужно найти элемент 85. Если взглянуть на диагональ, мы увидим элементы 35 и 95. Какую информацию о местонахождении элемента 85 можно из этого извлечь?

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

85 не может находиться в черной области, так как элемент 95 расположен в верхнем левом углу и является наименьшим элементом в этом квадрате.

85 не может принадлежать светло-серой области, так как элемент 35 находится в нижнем правом углу.

Следовательно, элемент 85 должен быть в одной из двух белых областей.

Таким образом, мы делим нашу сетку на четыре квадранта и выполняем поиск в нижнем левом и верхнем правом квадрантах. Их также можно разбить на квадранты и продолжить поиск.

Обратите внимание, что диагональ отсортирована, а значит, мы сможем эффективно использовать бинарный поиск.

Следующий код реализует этот алгоритм:

```
1  Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int x)
{
2    if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
3      return null;
4    }
5    if (matrix[origin.row][origin.column] == x) {
6      return origin;
7    } else if (!origin.isBefore(dest)) {
8      return null;
9    }
10
11    /* Start устанавливается в начало диагонали, а end - в конец.
12     * Если сетка не квадратная, end может отличаться от dest. */
13    Coordinate start = (Coordinate) origin.clone();
14    int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
15    Coordinate end = new Coordinate(start.row + diagDist, start.column +
                                      diagDist);
16    Coordinate p = new Coordinate(0, 0);
17
18    /* Провести бинарный поиск по диагонали и найти первый элемент > x */
19    while (start.isBefore(end)) {
20      p.setToAverage(start, end);
21      if (x > matrix[p.row][p.column]) {
22        start.row = p.row + 1;
23        start.column = p.column + 1;
24      } else {
25        end.row = p.row - 1;
26        end.column = p.column - 1;
27     }
28    }
29
30    /* Поиск ведется в левом нижнем и правом верхнем квадрантах сетки. */
31    return partitionAndSearch(matrix, origin, dest, start, x);
32  }
33
34  Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate
                                  dest,
35                                Coordinate pivot, int x) {
36    Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
37    Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
38    Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
```

```
39    Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);
40
41    Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
42    if (lowerLeft == null) {
43      return findElement(matrix, upperRightOrigin, upperRightDest, x);
44    }
45    return lowerLeft;
46  }
47
48  Coordinate findElement(int[][] matrix, int x) {
49    Coordinate origin = new Coordinate(0, 0);
50    Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
51    return findElement(matrix, origin, dest, x);
52  }
53
54  public class Coordinate implements Cloneable {
55    public int row, column;
56    public Coordinate(int r, int c) {
57      row = r;
58      column = c;
59    }
60
61    public boolean inbounds(int[][] matrix) {
62      return  row >= 0 && column >= 0 &&
63            row < matrix.length && column < matrix[0].length;
64    }
65
66    public boolean isBefore(Coordinate p) {
67      return row <= p.row && column <= p.column;
68    }
69
70    public Object clone() {
71      return new Coordinate(row, column);
72    }
73
74    public void setToAverage(Coordinate min, Coordinate max) {
75      row = (min.row + max.row) / 2;
76      column = (min.column + max.column) / 2;
77    }
78  }
```

Если вы посмотрели на этот код и подумали: «Я ни за что не напишу все это на собеседовании» — скорее всего, вы правы. Но ваши результаты будут сравнивать с работами других кандидатов. Если вы не смогли, они, вероятно, тоже не справятся. Не думайте, что вы находитесь в заведомо проигрышной ситуации, раз вам досталась такая сложная задача.

Вы немного облегчите себе жизнь, выделяя код в методы. Например, если выделить `partitionAndSearch` в отдельный метод, вам будет проще спланировать ключевые аспекты кода. Если у вас останется время, вы вернетесь к методу `partitionAndSearch` и напишете его тело.

10.10. Вы обрабатываете поток целых чисел. Периодически вам нужно находить ранг числа x (количество значений <= x). Какие структуры данных и алгоритмы необходимы для поддержки этих операций? Реализуйте метод `track(int x)`, вызываемый при генерировании каждого символа, и метод `getRankOfNumber(int x)`, возвращающий количество значений <=x (не включая x).

Пример:

```
Поток (в порядке появления): 5, 1, 4, 4, 5, 9, 7, 13, 3
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

РЕШЕНИЕ

Относительно простое решение — использовать массив, в котором все элементы хранятся в отсортированном порядке. Когда появляется новый элемент, ему понадобится место, а значит, придется сдвигать другие элементы. Реализация `getRankOfNumber` должна работать достаточно эффективно. Нам нужно выполнить бинарный поиск `n` и вернуть индекс.

Такое решение не очень эффективно в том, что касается вставки элементов (`track(int x)`). Нам нужна структура данных, которая хорошо подходит как для хранения элементов в отсортированном порядке, так и для вставки новых элементов. Бинарное дерево поиска соответствует этим требованиям.

Вместо того чтобы вставлять элементы в массив, можно вставлять элементы в бинарное дерево поиска. Метод `track(int x)` будет выполняться за время $O(\log n)$, где `n` — размер дерева (конечно, если дерево сбалансировано).

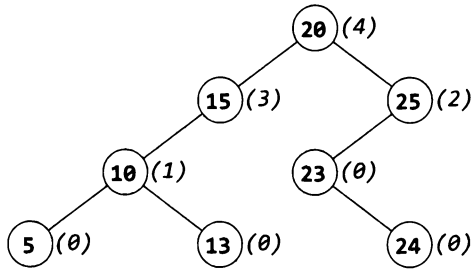
Чтобы найти позицию числа, можно выполнить симметричный обход, сохраняя при этом значение счетчика. К тому моменту, когда значение `x` будет обнаружено, счетчик сохранит информацию о количестве элементов, меньших `x`.

При движении влево во время поиска `x` счетчик (`counter`) не меняется. Почему? Потому что все значения, пропускаемые справа, больше `x`. Наименьший элемент (с рангом 1) является крайним левым узлом.

Когда мы двигаемся вправо, то проходим все левые элементы. Все эти элементы меньше `x`, поэтому нам нужно увеличить `counter` на количество элементов в левом поддереве.

Вместо того чтобы подсчитывать размер левого поддерева (что может быть неэффективно), можно отслеживать эту информацию при добавлении новых элементов в дерево.

Рассмотрим пример конкретного дерева. В следующем примере значения в скобках описывают количество узлов в левом поддереве (ранг узла *относительно* его поддерева).



Предположим, что мы хотим найти ранг значения 24 в этом дереве. Если сравнить 24 с корнем (20), становится очевидно, что значение 24 должно находиться правее. В левом поддереве есть четыре узла, таким образом, мы получаем пять узлов (включая корень), со значениями меньше 24. Таким образом, `counter=5`.

Затем можно сравнить 24 и 25; из этого следует, что значение 24 должно находиться левее. Значение `counter` не обновляется, поскольку мы «пропускаем» все меньшие узлы. Значение `counter` остается равным 5.

Теперь мы сравниваем 24 с 23 и понимаем, что значение 24 должно располагаться справа. Счетчик увеличивается на 1 (до 6), поскольку 23 не имеет «левых» узлов.

Наконец, мы находим значение 24 и возвращаем значение `counter=6`.

В рекурсивной реализации алгоритм выглядит так:

```
1  RankNode root = null;
2
3  void track(int number) {
4    if (root == null) {
5      root = new RankNode(number);
6    } else {
7      root.insert(number);
8    }
9  }
10
11  int getRankOfNumber(int number) {
12    return root.getRank(number);
13  }
14
15
16  public class RankNode {
17    public int left_size = 0;
18    public RankNode left, right;
19    public int data = 0;
20    public RankNode(int d) {
21      data = d;
22    }
23
24    public void insert(int d) {
25      if (d <= data) {
26        if (left != null) left.insert(d);
27        else left = new RankNode(d);
28        left_size++;
29      } else {
30        if (right != null) right.insert(d);
31        else right = new RankNode(d);
```

```
32      }
33    }
34
35    public int getRank(int d) {
36      if (d == data) {
37        return left_size;
38      } else if (d < data) {
39        if (left == null) return -1;
40        else return left.getRank(d);
41      } else {
42        int right_rank = right == null ? -1 : right.getRank(d);
43        if (right_rank == -1) return -1;
44        else return left_size + 1 + right_rank;
45      }
46    }
47  }
```

Оба метода — `track` и `getRankOfNumber` — выполняются за время $O(\log N)$ для сбалансированного дерева и за время $O(N)$ для несбалансированного дерева.

Обратите внимание на обработку ситуации, в которой значение `d` не найдено. Мы проверяем возвращаемое значение, и если оно равно -1, возвращаем -1 вверх по дереву. Очень важно помнить о подобных ситуациях на собеседовании.

10.11. Имеется массив целых чисел. Будем называть «пиком» элемент, больший соседних элементов либо равный им, а «впадиной» — элемент, меньший соседних элементов либо равный им. Например, в массиве {5, 8, 6, 2, 3, 4, 6} элементы {8, 6} являются пиками, а элементы {5, 2} — впадинами. Отсортируйте заданный массив целых чисел в чередующуюся последовательность пиков и впадин.

Пример:

Ввод: `{5, 3, 1, 2, 3}`

Вывод: `{5, 1, 3, 2, 3}`

РЕШЕНИЕ

Так как в этой задаче требуется отсортировать массив определенным образом, можно попытаться провести обычную сортировку, а затем «расставить» элементы массива в чередующуюся последовательность пиков и впадин.

Неоптимальное решение

Предположим, имеется отсортированный массив:

```
0  1  4  7  8  9
```

Этот список целых чисел упорядочен по возрастанию.

Как преобразовать его в чередующуюся последовательность пиков и впадин? Попробуем перебрать элементы массива и расставить их в нужном порядке.

- Элемент 0 оставляем на месте.

- Элемент расположен неправильно. Его можно поменять местами с 0 или 4; выбираем 0.

```
1  0  4  7  8  9
```

- Элемент 4 оставляем на месте.
- Элемент 7 расположен неправильно. Его можно поменять местами с 4 или 8; выбираем 4.

```
1  0  7  4  8  9
```

- Элемент 9 расположен неправильно. Поменяем его местами с 8.

```
1  0  7  4  9  8
```

Заметим, что в этих конкретных значениях нет ничего особенного. Важен относительный порядок элементов, но во всех отсортированных массивах он одинаков. Следовательно, этот способ применим для любого отсортированного массива.

Прежде чем браться за написание кода, следует формализовать алгоритм.

1. Отсортировать массив по возрастанию.
2. Перебрать элементы, начиная с индекса 1 (не 0), со смещением на два элемента.
3. Для каждого элемента поменять его местами с предыдущим элементом. Так как любые три элемента следуют в порядке *меньший* <= *средний* <= *больший*, в результате перестановки средний элемент всегда становится пиком: *средний* <= *меньший* <= *больший*.

Алгоритм гарантирует, что пики находятся на правильных местах: индексы 1, 3, 5 и т. д. Если элементы с нечетными номерами (пики) больше соседних элементов, то элементы с четными номерами (впадины) должны быть меньше своих соседних элементов.

Ниже приведена реализация алгоритма.

```
1  void sortValleyPeak(int[] array) {
2    Arrays.sort(array);
3    for (int i = 1; i < array.length; i += 2) {
4      swap(array, i - 1, i);
5    }
6  }
7
8  void swap(int[] array, int left, int right) {
9    int temp = array[left];
10    array[left] = array[right];
11    array[right] = temp;
12  }
```

Алгоритм выполняется за время $O(n \log n)$.

Оптимальное решение

Чтобы оптимизировать предыдущее решение, необходимо исключить этап сортировки. Алгоритм должен работать с несортированным массивом.

Вернемся к приведенному ранее примеру:

```
9  1  0  4  8  7
```

Для каждого элемента рассмотрим соседние элементы. Представим некоторые последовательности, в которых будут использоваться только числа 0, 1 и 2 (конкретные значения в данном случае роли не играют).

```
0  1  2
0  2  1    // пик
1  0  2
1  2  0    // пик
2  1  0
2  0  1
```

Если центральный элемент должен быть пиком, то подойдет любая из этих последовательностей. Можно ли «исправить» другие последовательности, чтобы сделать центральный элемент пиком?

Да, можно. Для этого следует поменять местами центральный элемент с наибольшим соседним элементом.

```
0  1  2  ->  0  2  1
0  2  1     // пик
1  0  2  ->  1  2  0
1  2  0     // пик
2  1  0  ->  1  2  0
2  0  1  ->  0  2  1
```

Как упоминалось ранее, если мы обеспечим правильное расположение пиков, то впадины также будут находиться в правильных местах.

Здесь необходима осторожность. Возможно ли, что одна из перестановок сможет нарушить уже обработанную часть последовательности? Хорошо, если вы об этом задумаетесь, но в данном случае такой проблемы нет. Если средний элемент меняется местами с левым, то левый в настоящее время является впадиной. Средний элемент меньше левого, поэтому мы размещаем еще меньший элемент как впадину. Ничто не нарушается, все хорошо.

Реализация приведена ниже.

```
1  void sortValleyPeak(int[] array) {
2    for (int i = 1; i < array.length; i += 2) {
3      int biggestIndex = maxIndex(array, i - 1, i, i + 1);
4      if (i != biggestIndex) {
5        swap(array, i, biggestIndex);
6      }
7    }
8  }
9
10 int maxIndex(int[] array, int a, int b, int c) {
11   int len = array.length;
12   int aValue = a >= 0 && a < len ? array[a] : Integer.MIN_VALUE;
13   int bValue = b >= 0 && b < len ? array[b] : Integer.MIN_VALUE;
14   int cValue = c >= 0 && c < len ? array[c] : Integer.MIN_VALUE;
15
16   int max = Math.max(aValue, Math.max(bValue, cValue));
17   if (aValue == max) return a;
18   else if (bValue == max) return b;
19   else return c;
20 }
```

Алгоритм выполняется за время $O(n)$.

11

Тестирование

11.1. Найдите ошибку (ошибки) в следующем коде:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3 printf("%d\n", i);
```

РЕШЕНИЕ

Код содержит две ошибки.

Во-первых, значения типа `unsigned int` по определению всегда больше либо равны нулю. Поэтому условие цикла `for` всегда будет истинно, и цикл будет выполняться бесконечно.

Правильный код для вывода всех чисел от 100 до 1 должен использовать условие `i > 0`. Если нам на самом деле нужно вывести нулевое значение, то следует добавить дополнительную команду `printf` после цикла `for`.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%d\n", i);
```

Во-вторых, вместо `%d` следует использовать `%u`, поскольку мы выводим `unsigned int`.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%u\n", i);
```

Этот код правильно выведет список чисел от 100 до 1 по убыванию.

11.2. Вам предоставили исходный код приложения, которое аварийно завершается после запуска. После десяти запусков в отладчике вы обнаруживаете, что каждый раз программа завершается в разных местах. Приложение однопоточное и использует только стандартную библиотеку C. Какие ошибки могут вызвать сбой приложения? Как вы организуете тестирование?

РЕШЕНИЕ

Вопрос в значительной степени зависит от типа диагностируемого приложения. Тем не менее ниже перечислены некоторые типичные причины случайных отказов.

1. *Случайная переменная*: приложение может использовать некоторое «случайное» значение или компонент, которые не имеют конкретного значения при каждом

выполнении программы. Примеры: данные, вводимые пользователем, сгенерированное случайное число, текущее время и т. д.

2. *Неинициализированная переменная*: приложение может использовать неинициализированную переменную, которая в некоторых языках программирования по умолчанию может принимать любое значение. Таким образом, код может каждый раз работать по-разному.
3. *Утечка памяти*: возможно, программа расходует все доступные ресурсы. Другие факторы абсолютно случайны, потому что они зависят от количества работающих в данный момент процессов. К этой же категории можно отнести переполнение кучи или повреждение данных в стеке.
4. *Внешние зависимости*: программа может зависеть от другого приложения, машины или ресурса. Если таких связей много, программа может «упасть» в любой момент.

Диагностика начинается со сбора как можно более подробной информации о приложении. Кто его запускает? Что оно делает? К какому типу относится приложение?

Хотя сбои происходят в разных местах, возможно, причина сбоя связана с конкретными компонентами или сценариями. Например, приложение может оставаться работоспособным, если его запустить и оставить без внимания, а сбой происходит только в определенный момент после загрузки файла. Или же сбой происходит при выполнении низкоуровневых операций, например при файловом вводе-выводе.

Диагностику удобно проводить методом исключения. Закройте все остальные приложения. Очень внимательно отслеживайте все свободные ресурсы. Если есть возможность отключить некоторые части программы, сделайте это. Запустите программу на другой машине и посмотрите, повторится ли ошибка. Чем больше факторов можно изменить (или исключить), тем проще найти проблему.

Кроме того, можно использовать специальные инструменты для проверки конкретных причин. Например, чтобы исследовать причину появления ошибок 2-го типа, можно использовать программы-анализаторы, выявляющие неинициализированные переменные.

Подобные задачи проверяют не только ваши умственные способности, но и стиль вашей работы. Вы постоянно перескакиваете с одного на другое и выдвигаете случайные предположения? Или вы подходите к решению задачи логически и организованно? Хотелось бы надеяться на последнее.

11.3. В компьютерной реализации шахмат имеется метод `boolean canMoveTo(int x, int y)`. Этот метод (часть класса `Piece`) возвращает результат, по которому можно понять, возможно ли перемещение фигуры на позицию `(x, y)`. Объясните, как вы будете тестировать данный метод.

РЕШЕНИЕ

К этой задаче можно применить два основных типа тестирования: проверка предельного случая (убедитесь, что в программе не будет сбоев при неправильном вводе) и тестирование общего случая. Начнем с первого.

Тип тестирования 1: проверка предельного случая

Нужно убедиться в том, что программа корректно обрабатывает ошибки ввода. Это означает, что нужно проверить следующие условия:

- Проверить ввод отрицательных значений `x` и `y`.
- Проверить, что произойдет, если `x` больше ширины доски.
- Проверить, что произойдет, если `y` больше высоты доски.
- Проверить, что произойдет, если доска будет полностью заполнена.
- Проверить, что произойдет, если доска пуста (или почти пуста).
- Проверить, что произойдет, если белых фигур больше, чем черных.
- Проверить, что произойдет, если черных фигур больше, чем белых.

Следует выяснить у интервьюера, как следует поступать в перечисленных случаях: возвращать `false` или генерировать исключение (и, соответственно, проводить тестирование).

Тип тестирования 2: общее тестирование

Общее тестирование — намного более объемная задача. В идеале нужно протестировать все возможные варианты расстановки фигур, но их слишком много. Значит, нужно проанализировать достаточное количество вариантов.

В шахматах используются 6 фигур, поэтому мы можем протестировать все варианты возможных расстановок пар фигур. Код будет выглядеть примерно так:

```
1 foreach фигура a:
2     для всех остальных типов фигур b (6 типов + пустая клетка)
3         foreach направление d
4             Создать доску с фигурой a.
5             Поместить фигуру b в направлении d.
6             Попытаться сделать ход - проверить возвращаемое значение.
```

Ключ к решению этой задачи — не пытаться протестировать все возможные сценарии. Вместо этого следует сосредоточиться на существенных областях.

11.4. Как вы проведете нагрузочный тест веб-страницы без использования специальных инструментов тестирования?

РЕШЕНИЕ

Нагрузочное тестирование помогает выявить максимальную операционную емкость веб-приложения и выявить любые «узкие места», которые могут оказать влияние на его производительность. Также оно показывает, как приложение реагирует на изменения нагрузки.

Чтобы провести нагрузочный тест, нужно сначала определить критические сценарии и метрики, по которым оцениваются цели производительности. Типичные критерии:

- время отклика;
- пропускная способность;
- использование ресурсов;
- максимальная нагрузка, которую может выдержать система.

Затем можно приступить к разработке тестов для моделирования нагрузки с учетом всех перечисленных критериев.

При отсутствии инструментов тестирования можно создать собственные. Например, чтобы смоделировать одновременную работу множества людей, можно создать тысячи виртуальных пользователей. Можно написать многопоточную программу с тысячами потоков, где каждый поток будет соответствовать реальному пользователю, загружающему страницу. Для каждого пользователя на программном уровне измеряются основные метрики: время отклика, скорость ввода-вывода и т. д.

Затем результаты, основанные на собранных данных, сравниваются с допустимыми показателями.

11.5. Как вы организуете тестирование авторучки?

РЕШЕНИЕ

Это задача на понимание ограничений и структурного подхода к решению.

Чтобы понять ограничения, необходимо задать много вопросов: «кто, что, где, когда, как и почему» (или по крайней мере те вопросы, которые актуальны для данной задачи). Не забывайте, что хороший тестер должен сначала понять, что тестирует, и только потом приступать к работе.

Чтобы проиллюстрировать решение этой задачи, давайте рассмотрим примерный диалог между интервьюером и кандидатом:

Интервьюер: Как бы вы протестировали авторучку?

Кандидат: Позвольте уточнить информацию о ручке. Кто собирается ее использовать?

Интервьюер: Вероятно, дети.

Кандидат: Хорошо, это интересно. Что они будут делать с ней? Они будут писать, рисовать или делать еще что-нибудь?

Интервьюер: Рисовать.

Кандидат: Хорошо. На чем? На бумаге? На ткани? На стене?

Интервьюер: На ткани.

Кандидат: Замечательно! О какой ручке идет речь? Фломастер, шариковая ручка? Должен рисунок отмываться или быть постоянным?

Интервьюер: Должен отмываться.

И вот вы, наконец, вы добираетесь до сути:

Кандидат: Хорошо, насколько я понял, ручка предназначена для детей 5–10 лет. Это фломастер, который может быть красного, зеленого, синего или черного цвета. Рисунок на ткани должен отмываться. Это правильно?

Теперь у кандидата гораздо больше данных, чем было изначально. Многие интервьюеры преднамеренно дают задачу, которая кажется простой и понятной (ведь все знают, что такое ручка), только для того, чтобы вы выяснили, что задача сильно отличается от исходных предположений. Они считают, что пользователи делают то же самое, но делают это случайно.

Теперь, когда вы понимаете, что́ тестируете, можно подумать над решением. Ключ к нему — структура. В этом случае компонентами могут быть:

- *Проверка факта*: удостовериться, что ручка является фломастером одного из разрешенных цветов.
- *Намеченное использование*: рисование. Ручка оставляет следы на ткани?
- *Намеченное использование*: стирка. Чернила отстирываются? (Даже если рисунок был сделан давно?) В какой воде они отстирываются — горячей, теплой, холодной?
- *Безопасность*: действительно ли ручка безопасна (нетоксична) для детей?
- *Непреднамеренное использование*: как еще дети могут использовать ручку? Они могут рисовать на других поверхностях, поэтому вы должны проверить, пишет ли на них ручка. Они могут также уронить ручку на пол, встать на нее, бросить и т. д. Нужно убедиться, что ручка не сломается в таких обстоятельствах.

Помните, что в любой задаче на тестирование необходимо проверить использование предмета как по назначению, так и не по назначению. Люди не всегда используют вещи так, как вам хочется.

11.6. Как вы организуете тестирование банкомата, подключенного к распределенной банковской системе?

РЕШЕНИЕ

Первое, что нужно сделать, — уточнить исходные предположения. Задайте следующие вопросы:

- Кто будет использовать банкомат? Вам могут ответить, например, «кто угодно» или «только слепые люди» или еще что-нибудь.
- Для чего будут использовать банкомат? Возможные ответы: снимать деньги, оплачивать услуги, проверять баланс и т. д.
- Какие инструменты доступны для тестирования? Специальное программное обеспечение или только сам банкомат?

Помните: хороший тестер должен знать, что он тестирует!

Как только вы начинаете представлять, что собой представляет система, задачу можно разбить на компоненты для раздельного тестирования. К числу таких компонентов относятся:

- ввод регистрационных данных пользователя;
- снятие средств;
- внесение средств;
- проверка баланса;
- перевод денег.

Вероятно, следует предусмотреть как ручное, так и автоматизированное тестирование.

Ручное тестирование позволяет провести все перечисленные операции и проверить все возможные ошибки (низкий баланс, новый счет, несуществующий счет и т. д.).

С автоматизированным тестированием дело обстоит сложнее. Мы автоматически тестируем все стандартные сценарии и проверяем некоторые специфические случаи (например, «ситуацию гонки»). В идеале желательно настроить закрытую систему с фиктивными счетами и убедиться, что даже если кто-нибудь попытается быстро вносить или снимать деньги из разных мест, он никогда не получит лишнего и не потеряет своего.

На первое место ставятся безопасность и надежность. Счета клиентов должны быть защищены, и мы должны обеспечить полную подотчетность всех средств. Никто не хочет неожиданно потерять свои деньги! Хороший тестер понимает приоритеты той системы, которую он тестирует.

12

C и C++

12.1. Напишите на C++ метод для вывода последних K строк входного файла.

РЕШЕНИЕ

Одно из решений методом «грубой силы» подсчитывает количество строк (N) и выводит строки с N–K до N. Но для этого понадобится дважды прочитать файл, что требует лишнего времени. Поищем решение, которое читает файл только один раз и выводит последние K строк.

Можно создать массив для K строк и прочитать последние K строк. Каждый раз при чтении новой строки мы будем удалять самую старую строку из массива.

Возникает резонный вопрос: разве может быть эффективным решение, требующее постоянного сдвига элементов в массиве? Это решение станет эффективным, если мы правильно реализуем сдвиг. Вместо того чтобы каждый раз выполнять сдвиг массива, можно воспользоваться циклическим массивом.

В циклическом массиве при чтении новой строки всегда будет заменяться самый старый элемент. Для его отслеживания используется отдельная переменная, которая будет меняться при добавлении новых элементов.

Пример использования циклического массива:

```
шаг 1 (исходное состояние): массив = {a, b, c, d, e, f}. p = 0
шаг 2 (вставка g):  массив = {g, b, c, d, e, f}. p = 1
шаг 3 (вставка h):  массив = {g, h, c, d, e, f}. p = 2
шаг 4 (вставка i):  массив = {g, h, i, d, e, f}. p = 3
```

Ниже приведена реализация этого алгоритма.

```
1  void printLast10Lines(char* fileName) {
2    const int K = 10;
3    ifstream file (fileName);
4    string L[K];
5    int size = 0;
6
7    /* Файл читается по строкам в циклический массив. Используем peek(), */
8    /* чтобы маркер EOF за концом строки не считался отдельной строкой */
9    while (file.peek() != EOF) {
10     getline(file, L[size % K]);
11     size++;
12   }
13
14   /* Вычисление начала циклического массива и его размера */
15   int start = size > K ? (size % K) : 0;
```

```
16    int count = min(K, size);
17
18    /* Вывод элементов в порядке их чтения */
19    for (int i = 0; i < count; i++) {
20      cout << L[(start + i) % K] << endl;
21    }
22  }
```

В этом решении читается весь файл, но в памяти в любой момент времени хранятся только 10 строк.

12.2. Реализуйте на C или C++ функцию `void reverse(char* str)` для перестановки символов строки, завершенной нуль-символом, в обратном порядке.

РЕШЕНИЕ

Это весьма типичный вопрос для собеседования. Единственная потенциальная проблема связана с выполнением операции «на месте». Также следует проявить внимательность при обработке символа `null`.

Реализуем эту функцию на C.

```
1  void reverse(char *str) {
2    char* end = str;
3    char tmp;
4    if (str) {
5      while (*end) { /* Найти конец строки */
6        ++end;
7     }
8      --end; /* Отступить на 1 символ, так как строка завершается null */
9
10     /* Меняем местами символы в начале и конце строки, пока
11      * указатели не встретятся в середине. */
12     while (str < end) {
13       tmp = *str;
14       *str++ = *end;
15       *end-- = tmp;
16    }
17   }
18 }
```

Существует много способов решения этой задачи. Также возможна рекурсивная реализация (хотя выбирать ее не рекомендуется).

12.3. Проведите сравнительный анализ хеш-таблицы и ассоциативного массива из стандартной библиотеки шаблонов (STL). Как реализована хеш-таблица? Если объем данных невелик, какие структуры данных можно использовать вместо хеш-таблицы?

РЕШЕНИЕ

В хеш-таблицу значение попадает в результате вызова хеш-функции с ключом. Сами значения хранятся в несортированном порядке. Так как хеш-таблица использует ключ для индексации элементов, вставка или поиск данных занимает амортизированное время $O(1)$ (при небольшом количестве коллизий в хеш-таблицах).

В хеш-таблице также нужно обрабатывать потенциальные коллизии. Для этого в каждом индексе сохраняется связный список всех значений, ключи которых отображаются на конкретный индекс.

Контейнер STL `map` вставляет пары «ключ/значение» в дерево бинарного поиска по ключу. При этом не требуется обрабатывать коллизии, а так как дерево сбалансировано, время вставки и поиска составляет $O(\log N)$.

Как реализована хеш-таблица?

Хеш-таблица реализуется как массив связных списков. При вставке пары «ключ — значение» хеш-функция отображает ключ на индекс массива. Значение вставляется в связный список в полученную позицию.

Было бы неправильно утверждать, что элементы связного списка с определенным индексом массива соответствуют одному ключу. Просто функция `hashFunction(key)` возвращает для этих ключей одинаковые результаты. Поэтому для получения значения, соответствующего ключу, в каждом узле необходимо хранить и ключ, и значение.

Подведем итог: хеш-таблица реализуется как массив связных списков, где каждый узел списка содержит два компонента: значение и исходный ключ. Основные особенности реализации хеш-таблиц:

1. Необходимо выбрать хеш-функцию, которая обеспечит хорошее распределение ключей. Если ключи будут распределены неравномерно, то возникнет множество коллизий, и скорость нахождения элемента снизится.
2. Какой бы хорошей ни была хеш-функция, коллизии все равно будут возникать, и их нужно будет обрабатывать. Это подразумевает использование связных списков (или другой метод решения проблемы).
3. Можно реализовать методы динамического увеличения или уменьшения размера хеш-таблицы. Например, когда отношение количества элементов к размеру таблицы превышает определенное значение, хеш-таблица увеличивается. Это означает, что нам потребуется создать новую хеш-таблицу и передать в нее записи из старой. Поскольку этот процесс сопряжен с высокими затратами, нужно сделать все возможное, чтобы размер таблицы не менялся слишком часто.

Чем можно заменить хеш-таблицу при небольшом объеме входных данных?

Используйте контейнер `map` из STL или бинарное дерево. Хотя это потребует времени $O(\log(n))$, объем данных невелик, поэтому временные затраты будут незначительными.

12.4. Как работают виртуальные функции в C++?

РЕШЕНИЕ

Виртуальные функции работают на основе *v-таблиц* (vtables). Если какая-либо функция класса объявлена как виртуальная, создается v-таблица с адресами виртуальных функций этого класса. Для всех таких классов компилятор добавляет

скрытую переменную `vptr`, которая содержит указатель на v-таблицу. Если виртуальная функция не переопределена в производном классе, v-таблица производного класса хранит адрес функции из родительского класса. V-таблица используется для получения адреса функции при вызове виртуальной функции. Механизм v-таблиц позволяет реализовать динамическое связывание в С++.

Когда указателю на базовый класс присваивается объект производного класса, переменная `vptr` указывает на v-таблицу производного класса. Присваивание гарантирует, что будет вызвана виртуальная функция производного класса с максимальной глубиной наследования.

Рассмотрим следующий код:

```
1  class Shape {
2    public:
3    int edge_length;
4    virtual int circumference () {
5      cout << "Circumference of Base Class\n";
6      return 0;
7    }
8  };
9
10 class Triangle: public Shape {
11   public:
12   int circumference () {
13     cout<< "Circumference of Triangle Class\n";
14     return 3 * edge_length;
15   }
16 };
17
18 void main() {
19   Shape * x = new Shape();
20   x->circumference(); // "Circumference of Base Class"
21   Shape *y = new Triangle();
22   y->circumference(); // "Circumference of Triangle Class"
23 }
```

В предыдущем примере функция `circumference` — виртуальная функция класса `Shape` — является виртуальной в каждом из произвольных классов (`Triangle` и т. д.). В С++ вызовы невиртуальных функций обрабатываются во время компиляции с применением статического связывания, а вызовы виртуальной функции обрабатываются во время выполнения с применением динамического связывания.

12.5. Чем глубокое копирование отличается от поверхностного? Объясните, как использовать эти виды копирования.

РЕШЕНИЕ

При поверхностном копировании все значения членов класса копируются из одного объекта в другой. При глубоком копировании происходит то же самое, но также копируются все объекты, на которые ссылаются указатели.

Пример поверхностного и глубокого копирования:

```
1  struct Test {
2    char * ptr;
3  };
4
5  void shallow_copy(Test & src, Test & dest) {
6    dest.ptr = src.ptr;
7  }
8
9  void deep_copy(Test & src, Test & dest) {
10   dest.ptr = (char*)malloc(strlen(src.ptr) + 1);
11   strcpy(dest.ptr, src.ptr);
12 }
```

Функция `shallow_copy` может породить немало ошибок времени исполнения, особенно при создании и удалении объектов. Эту функцию нужно использовать очень осторожно и только когда программист действительно знает, что делает. В большинстве случаев поверхностное копирование используют, когда нужно передать информацию о сложной структуре без дублирования данных. Будьте осторожны! Нужно очень внимательно обходиться с «разрушением» объектов при поверхностном копировании.

На практике поверхностное копирование применяется редко. В большинстве случаев используется глубокое копирование, особенно если размер копируемой структуры относительно невелик.

12.6. Каково назначение ключевого слова volatile в C?

РЕШЕНИЕ

Ключевое слово `volatile` сообщает компилятору, что значение переменной может меняться под влиянием внешних обстоятельств, то есть без явного обновления из программного кода. Например, это может происходить под управлением операционной системы, аппаратных средств или другого потока. Поскольку значение может неожиданно измениться, компилятор каждый раз загружает его из памяти.

Нестабильную целочисленную переменную можно объявить как:

```
int volatile x;
volatile int x;
```

Указатель на нестабильную переменную объявляется следующим образом:

```
volatile int * x;
int volatile * x;
```

Нестабильный указатель на стабильные данные теоретически возможен, но встречается редко:

```
int * volatile x;
```

Если вы хотите объявить нестабильный указатель на нестабильную область памяти, необходимо сделать следующее:

```
int volatile * volatile x;
```

Нестабильные переменные не оптимизируются компилятором, что может быть очень полезно. Представьте следующую функцию:

```
1  int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

На первый взгляд этот код будет выполняться бесконечно. Компилятор может попытаться оптимизировать его и привести к следующему виду:

```
1  void Fn(void) {
2    start:
3      int opt = 1;
4      if (true)
5      goto start;
6  }
```

Теперь цикл точно становится бесконечным. Однако внешняя операция могла записать `0` в переменную `opt` и прервать цикл.

Чтобы предотвратить такую оптимизацию, можно объявить, что переменная может измениться под воздействием некоего внешнего фактора. Для этого используется ключевое слово `volatile`:

```
1  volatile int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

Нестабильные переменные также используются в качестве глобальных в многопотоковых программах, в которых они могут быть изменены любым потоком. Операции с такими переменными не должны оптимизироваться компилятором.

12.7. Почему деструктор базового класса должен объявляться виртуальным?

РЕШЕНИЕ

Подумаем, для чего вообще нужны виртуальные методы. Предположим, имеется следующий код:

```
1  class Foo {
2   public:
3    void f();
4  };
5
6  class Bar : public Foo {
7   public:
8    void f();
9  }
10
11  Foo * p = new Bar();
12  p->f();
```

Вызов `p->f()` приводит к вызову `Foo::f()`. Это объясняется тем, что `p` — это указатель на `Foo`, а функция `f()` не является виртуальной.

Чтобы при вызове `p->f()` была вызвана реализация `f()` с максимальным уровнем наследования, необходимо объявить функцию `f()` как виртуальную.

Теперь вернемся к деструктору. Деструкторы предназначены для очистки памяти и ресурсов. Если деструктор `Foo` не является виртуальным, то даже при уничтожении объекта с *фактическим* типом `Bar` все равно будет вызван деструктор базового класса `Foo`.

Поэтому деструкторы объявляют виртуальными — это гарантирует, что будет вызван деструктор производного класса с максимальным уровнем наследования.

12.8. Напишите метод, получающий указатель на структуру Node в параметре и возвращающий полную копию переданной структуры данных. Структура данных Node содержит два указателя на другие узлы (другие структуры Nodes).

РЕШЕНИЕ

Алгоритм будет хранить информацию отображения адресов узлов исходной структуры на соответствующие узлы новой структуры. Эта связь позволит обнаруживать узлы, которые были скопированы ранее во время традиционного обхода структуры в глубину. При обходе посещенные узлы часто помечаются, но пометка может существовать в самых разных формах, и ее не обязательно хранить в узле.

Таким образом, мы получаем достаточно простой рекурсивный алгоритм:

```
typedef map<Node*, Node*> NodeMap;

Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
  if (cur == NULL) {
    return NULL;
  }

  NodeMap::iterator i = nodeMap.find(cur);
  if (i != nodeMap.end()) {
     // Узел уже встречался, вернуть копию
     return i->second;
  }

  Node * node = new Node;
  nodeMap[cur] = node; // Сохранить текущий узел перед обходом ссылок
  node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
  node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
  return node;
}

Node * copy_structure(Node * root) {
  NodeMap nodeMap; // Пустой контейнер
  return copy_recursive(root, nodeMap);
}
```

12.9. Напишите класс «умного указателя» — типа данных (обычно реализованного на базе шаблона), моделирующего указатель с поддержкой автоматической уборки мусора. Умный указатель автоматически подсчитывает количество ссылок на объект `SmartPointer<T*>` и освобождает объект типа `T`, когда число ссылок становится равным 0.

РЕШЕНИЕ

Умный указатель представляет собой обычный указатель с поддержкой автоматического управления памятью. Такой указатель помогает избежать многих проблем: «висячих» указателей, утечки памяти и неудачных попыток выделения памяти. Умный указатель должен подсчитывать количество ссылок на указанный объект.

На первый взгляд эта задача кажется довольно сложной, особенно если вы не эксперт в C++. Один из полезных приемов для ее решения — разделить задачу на две части: 1) составить общую схему решения и написать псевдокод, а затем 2) реализовать полноценный код.

Нам понадобится переменная — счетчик ссылок, которая будет увеличиваться при создании новой ссылки на объект и уменьшаться при удалении ссылки. Псевдокод будет выглядеть примерно так:

```
1 template <class T> class SmartPointer {
2    /* Класс умного указателя должен содержать указатели на сам объект
3    * и на счетчик ссылок. Оба они должны быть указателями (а не реальным
4    * объектом или значением счетчика), так как суть умного указателя
5    * заключается в использовании счетчика ссылок в нескольких
6    * экземплярах умного указателя на один объект  */
7     T * obj;
8     unsigned * ref_count;
9 }
```

Класс должен содержать конструктор и деструктор, начнем с их добавления:

```
1 SmartPointer(T * object) {
2     /* Задать T * obj и инициализировать счетчик ссылок в 1. */
3 }
4
5 SmartPointer(SmarterPointer<T>& sptr) {
6     /* Конструктор создает новый умный указатель для существующего
7      * объекта. Сначала нужно присвоить obj и ref_count указатели
8      * на obj и ref_count объекта sptr. Затем, поскольку создается новая
9      * ссылка на obj, необходимо увеличить ref_count. */
10 }
11
12 ~SmartPointer(SmarterPointer<T> sptr) {
13     /* Уничтожение ссылки на объект. Уменьшаем ref_count.
14      *  Если ref_count = 0, освободить память и уничтожить
15      * объект. */
16 }
```

Ссылки также могут создаваться присваиванием объекту `SmartPointer` другого объекта `SmartPointer`. Для обработки этого случая нужно переопределить оператор = , но сначала набросаем эскиз кода:

```
1  onSetEquals(SmartPointer<T> ptr1, SmartPointer<T> ptr2) {
2      /* Если ptr1 имеет существующее значение, уменьшить счетчик ссылок.
3     * Затем скопировать указатели на obj и ref_count. Наконец,
4     * так как мы создали новую ссылку, увеличить ref_count. */
5 }
```

Собственно, даже без сложного синтаксиса C++ дело наполовину сделано. Осталось только заполнить недостающие подробности:

```
template <class T> class SmartPointer {
 public:
  SmartPointer(T * ptr) {
    ref = ptr;
    ref_count = (unsigned*)malloc(sizeof(unsigned));
    *ref_count = 1;
  }

  SmartPointer(SmartPointer<T> & sptr) {
     ref = sptr.ref;
     ref_count = sptr.ref_count;
     ++(*ref_count);
   }

   /* Переопределение оператора присваивания, чтобы при присваивании
    * одного умного указателя другому происходило обновление
    * счетчика ссылок старого указателя. */
   SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
     if (this == &sptr) return *this;

     /* Если счетчик ссылок положителен, удалить одну ссылку. */
     if (*ref_count > 0) {
       remove();
     }

     ref = sptr.ref;
     ref_count = sptr.ref_count;
     ++(*ref_count);
     return *this;
   }

   ~SmartPointer() {
     remove(); // Удалить одну ссылку на объект.
   }

   T getValue() {
     return *ref;
   }

  protected:
   void remove() {
     --(*ref_count);
     if (*ref_count == 0) {
       delete ref;
       free(ref_count);
       ref = NULL;
       ref_count = NULL;
     }
   }

   T * ref;
   unsigned * ref_count;
};
```

Код получается достаточно сложным. Вряд ли интервьюер ожидает, что вы напишете его без единой ошибки.

12.10. Напишите функции динамического выделения и освобождения памяти, которые работают с памятью таким образом, что возвращаемый адрес памяти кратен степени 2.

Пример:

`align_malloc(1000,128)` возвращает адрес памяти, который является кратным 128 и указывает на блок памяти размером 1000 байт.

Функция `aligned_free()` освобождает память, выделенную с помощью `align_malloc`.

РЕШЕНИЕ

Как правило, при вызове `malloc` пользователь не может управлять тем, где именно в пределах кучи будет выделена память. Мы просто получаем указатель на блок памяти, который может начинаться с любого адреса памяти в куче.

С учетом ограничений нам придется запрашивать достаточный объем памяти, чтобы мы могли вернуть адрес памяти, кратный нужной величине.

Предположим, нам нужен 100-байтовый блок и мы хотим, чтобы он начинался с адреса, кратного 16. Сколько памяти необходимо выделить, чтобы гарантировать выполнение условия? Для этого понадобятся 15 дополнительных байтов. С этими 15 байтами и 100 байтами, следующими за ними, в выделенной памяти заведомо найдется адрес 100-байтового блока, кратный 16.

Реализация выглядит примерно так:

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    int offset = alignment - 1;
3    void* p = (void*) malloc(required_bytes + offset);
4    void* q = (void*) (((size_t)(p) + offset) & ~(alignment - 1));
5    return q;
6  }
```

Строка 4 весьма непростая, и о ней стоит поговорить подробнее. Допустим, `aligment = 16`. Мы знаем, что где-то в первых 16 байтах блока по адресу `p` существует адрес памяти, кратный 16. Выражение `(p+15) & 11...10000` перемещает нас к этому адресу. Операция `AND` последних четырех битов `p+15` с `0000` гарантирует, что новое значение будет кратно 16 (либо по исходному значению `p`, либо по одному из следующих 15 адресов).

Решение *почти* идеально, за исключением одного: как освобождать память?

В рассмотренном примере мы выделили лишние 15 байтов и должны освободить их, когда будем освобождать «реальную» память.

Для этого можно сохранить в «дополнительной» памяти адрес начала полного блока памяти. Сохраним эту информацию перед «выровненным» блоком памяти. Конечно, это означает, что нам придется выделять *больше* дополнительной памяти, чтобы получить достаточно места для хранения указателя.

Следовательно, чтобы обеспечить как выравнивание адреса, так и хранение указателя, требуется дополнительно выделить `alignment - 1 + sizeof(void*)` байтов.

Приведенный далее код реализует это решение:

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    void* p1; // Исходный блок
3    void* p2; // Выровненный блок внутри исходного
4    int offset = alignment - 1 + sizeof(void*);
5    if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
6      return NULL;
7    }
8    p2 = (void*)(((size_t)(p1) + offset) & ~(alignment - 1));
9    ((void **)p2)[-1] = p1;
10   return p2;
11 }
12
13 void aligned_free(void *p2) {
14   /* Для единообразия используем те же имена, что и в aligned_malloc*/
15   void* p1 = ((void**)p2)[-1];
16   free(p1);
17 }
```

Обратите внимание на математические операции с указателями в строках 9 и 15. Если рассматривать `p2` как `void**` (или как массив элементов `void*`), для получения `p1` достаточно обратиться к индексу –1.

Метод `aligned_free` получает `p2` (то же значение `p2`, которое возвращал метод `aligned_malloc`). Как и прежде, значение `p1` (которое указывает на начало полного блока памяти) хранится непосредственно перед `p2`. Освобождая `p1`, мы освобождаем весь блок памяти.

12.11. Напишите на C функцию (`my2DAlloc`), которая выделяет память для двумерного массива. Минимизируйте количество вызовов `malloc` и примите меры к тому, чтобы для обращения к памяти можно было использовать синтаксис `arr[i][j]`.

РЕШЕНИЕ

Как известно, двумерный массив — это фактически массив массивов. Так как мы используем указатели с массивами, для создания двойного массива можно использовать двойные указатели.

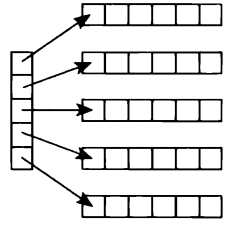
Основная идея заключается в создании одномерного массива указателей. Затем для каждого индекса в массиве создается новый одномерный массив. Таким образом мы получаем двумерный массив, к которому можно получить доступ по индексам массива.

Ниже приведена реализация этой идеи:

```
1  int** my2DAlloc(int rows, int cols) {
2    int** rowptr;
3    int i;
4    rowptr = (int**) malloc(rows * sizeof(int*));
5    for (i = 0; i < rows; i++)  {
6       rowptr[i] = (int*) malloc(cols * sizeof(int));
```

```
7    }
8     return rowptr;
9  }
```

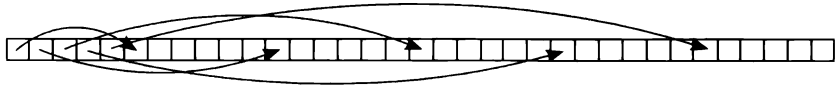
В этом коде мы сообщаем `rowptr`, на что именно должен указывать каждый элемент. Схема выделения памяти представлена на следующей диаграмме.



Чтобы освободить память, вызвать `free` для `rowptr` будет недостаточно. Необходимо проследить за тем, чтобы освобождалась не только память, выделенная первым вызовом `malloc`, но и память, выделенная всеми последующими вызовами.

```
1  void my2DDealloc(int** rowptr, int rows) {
2    for (i = 0; i < rows; i++) {
3       free(rowptr[i]);
4    }
5    free(rowptr);
6  }
```

Вместо того чтобы выделять память в разных блоках (по блоку для каждого ряда плюс блок для указания того, *где* расположен каждый ряд), можно выделить один смежный блок памяти. Концептуально для двумерного массива из 5 строк и 6 столбцов такой блок выглядит так:



Если подобная структура двумерного массива кажется вам странной, вспомните, что такое решение принципиально не отличается от первой схемы. Единственное отличие заключается в том, что память занимает смежный блок, поэтому первые пять (в данном примере) элементов указывают на разные места в этом блоке.

Реализация выглядит так:

```
1  int** my2DAlloc(int rows, int cols) {
2    int i;
3    int header = rows * sizeof(int*);
4    int data = rows * cols * sizeof(int);
5    int** rowptr = (int**)malloc(header + data);
6    if (rowptr == NULL) return NULL;
7
8    int* buf = (int*) (rowptr + rows);
9    for (i = 0; i < rows; i++) {
10      rowptr[i] = buf + i * cols;
```

```
11    }
12    return rowptr;
13  }
```

Обратите особое внимание на строки 11–13. Если массиво содержит 5 строк по 6 столбцов каждая, `array[0]` будет указывать на `array[5]`, `array[1]` — на `array[11]` и т. д.

Затем при вызове `array[1][3]` компьютер отыщет указатель `array[1]`, который указывает на другое место в памяти — `array[5]`. Этот элемент рассматривается как самостоятельный массив, после чего из него извлекается третий элемент (индексы начинаются с 0).

К преимуществам построения массива одним вызовом `malloc` относится и простота освобождения памяти. Теперь для этого достаточно одного вызов `free` без отдельных вызовов для освобождения остальных блоков данных.

13

Java

13.1. Как повлияет на наследование объявление конструктора приватным?

РЕШЕНИЕ

Объявление конструктора класса `A` со спецификатором `private` гарантирует, что конструктор будет доступен только для того, для кого доступны приватные методы `A`. Кто еще, кроме `A`, может обращаться к приватным методам и конструкторам `A`? Внутренние классы `A`. Кроме того, если `A` содержит внутренний класс `Q`, то и внутренние классы `Q` тоже смогут к ним обращаться.

Все это напрямую отражается на наследовании, потому что субкласс вызывает конструктор своего родителя. Класс `A` может использоваться для наследования, но только внутренними классами (его собственными или родительскими).

13.2. Будет ли выполняться блок `finally` (в коде Java), если оператор `return` находится внутри `try`-блока (конструкция `try-catch-finally`)?

РЕШЕНИЕ

Да, будет. Блок `finally` выполняется при выходе из блока `try`. Даже при попытке принудительного выхода из блока `try` (командой `return`, `continue`, `break` или еще каким-либо образом), блок `finally` все равно будет выполнен.

Блок `finally` не выполняется только в экстренных случаях, например:

- виртуальная машина прекратила свою работу во время выполнения блока `try/catch`;
- поток, который выполнял блок `try/catch`, был уничтожен.

13.3. В чем разница между `final`, `finally` и `finalize`?

РЕШЕНИЕ

Несмотря на похожие названия, `final`, `finally` и `finalize` имеют разные предназначения. `final` управляет «изменяемостью» переменной, метода или класса. `finally` используется в конструкции `try/catch`, чтобы гарантировать выполнение сегмента кода. Метод `finalize()` вызывается сборщиком мусора, как только последний решает, что ссылок больше не существует.

Ниже эти ключевые слова рассматриваются более подробно.

final

Назначение команды `final` может меняться в зависимости от контекста:

- С переменной (примитив): значение переменной не может изменяться.
- С переменной (ссылка): переменная не может указывать ни на какой другой объект в куче.
- С методом: метод не может переопределяться.
- С классом: класс не может субклассироваться.

finally

Необязательный блок `finally` может располагаться после блоков `try` или `catch`. Команды, находящиеся в блоке `finally`, выполняются всегда (исключение — завершение работы виртуальной машины Java в блоке `try`). Блок `finally` предназначен для выполнения завершающих действий, то есть «зачистки». Он выполняется после блоков `try` и `catch`, но до возврата управления к исходной точке.

Следующий пример показывает, как это делается:

```
1  public static String lem() {
2    System.out.println("lem");
3    return "return from lem";
4  }
5
6  public static String foo() {
7    int x = 0;
8    int y = 5;
9    try {
10     System.out.println("start try");
11     int b = y / x;
12     System.out.println("end try");
13     return "returned from try";
14   } catch (Exception ex) {
15     System.out.println("catch");
16     return lem() + " | returned from catch";
17   } finally {
18     System.out.println("finally");
19   }
20 }
21
22 public static void bar() {
23   System.out.println("start bar");
24   String v = foo();
25   System.out.println(v);
26   System.out.println("end bar");
27 }
28
29 public static void main(String[] args) {
30   bar();
31 }
```

Результат выполнения этого кода выглядит так:

```
1  start bar
2  start try
```

```
3  catch
4  lem
5  finally
6  return from lem | returned from catch
7  end bar
```

Присмотритесь к строкам 3–5. Блок `catch` отрабатывает полностью (включая вызов функции в команде `return`), затем выполняется блок `finally`, и только после этого функция возвращает управление.

finalize()

Метод `finalize()` вызывается сборщиком мусора непосредственно перед уничтожением объекта. Таким образом, класс может переопределить метод `finalize()` из класса `Object` для реализации нестандартного поведения в процессе уборки мусора.

```
1  protected void finalize() throws Throwable {
2    /* Закрыть открытые файлы, освободить ресурсы и т. д. */
3  }
```

13.4. Объясните разницу между шаблонами C++ и обобщениями (generics) в языке Java.

РЕШЕНИЕ

Многие программисты полагают, что шаблон и обобщения — это одно и то же: ведь в обоих случаях можно использовать конструкции вида `List<String>`. Но то, *как* и *почему* каждый из языков достигает нужного эффекта, существенно различается.

Обобщения Java происходят от идеи «стирания типа». Этот метод устраняет параметризацию типов при преобразовании исходного кода в байт-код JVM.

Допустим, имеется следующий Java-код:

```
1 Vector<String> vector = new Vector<String>();
2 vector.add(new String("hello"));
3 String str = vector.get(0);
```

Во время компиляции он будет преобразован к виду:

```
1 Vector vector = new Vector();
2 vector.add(new String("hello"));
3 String str = (String) vector.get(0);
```

Использование обобщений Java практически не повлияло на функциональность, но сделало код более красивым. Поэтому обобщения Java часто называют «синтаксическим украшением».

С шаблонами C++ дело обстоит совершенно иначе. Шаблоны в C++ по сути представляют собой набор макросов, создающих новую копию шаблонного кода для каждого типа. Это убедительно доказывает тот факт, что экземпляр `MyClass<Foo>` не будет использовать статическую переменную совместно с экземпляром `MyClass<Bar>`. С другой стороны, два экземпляра `MyClass<Foo>` будут совместно использовать статическую переменную.

Чтобы проиллюстрировать этот пример, рассмотрим следующий код:

```
1  /*** MyClass.h ***/
2  template<class T> class MyClass {
3   public:
4    static int val;
5    MyClass(int v) { val = v; }
6  };
7
8  /*** MyClass.cpp ***/
9  template<typename T>
10  int MyClass<T>::bar;
11
12  template class MyClass<Foo>;
13  template class MyClass<Bar>;
14
15  /*** main.cpp ***/
16  MyClass<Foo> * foo1 = new MyClass<Foo>(10);
17  MyClass<Foo> * foo2 = new MyClass<Foo>(15);
18  MyClass<Bar> * bar1 = new MyClass<Bar>(20);
19  MyClass<Bar> * bar2 = new MyClass<Bar>(35);
20
21  int f1 = foo1->val; // будет равно 15
22  int f2 = foo2->val; // будет равно 15
23  int b1 = bar1->val; // будет равно 35
24  int b2 = bar2->val; // будет равно 35
```

В Java статические переменные совместно используются экземплярами `MyClass` независимо от параметров-типов.

У обобщений Java и шаблонов C++ есть и другие отличия:

- Шаблоны C++ могут использовать примитивные типы (например, `int`). Для обобщений Java это невозможно, они обязаны использовать `Integer`.
- Java позволяет ограничить параметры обобщения определенным типом. Например, вы можете использовать обобщения для реализации `CardDeck` и указать, что параметр-тип должен расширять `CardGame`.
- В C++ можно создать экземпляр параметра-типа, в Java такая возможность отсутствует.
- В Java параметр-тип (`Foo` в `MyClass<foo>`) не может использоваться для статических методов и переменных, так как это привело бы к их совместному использованию `MyClass<Foo>` и `MyClass<Bar>`. В C++ это разные классы, поэтому параметр-тип может использоваться для статических методов и переменных.
- В Java все экземпляры `MyClass` независимо от их типизированных параметров относятся к одному и тому же типу. Параметры-типы «стираются» во время выполнения. В C++ экземпляры с разными параметрами-типами относятся к разным типам.

Помните, что хотя обобщения Java и шаблоны C++ внешне похожи, у них много различий.

13.5. Объясните различия между `TreeMap`, `HashMap` и `LinkedHashMap`. Приведите пример ситуации, в которой каждая из этих коллекций работает лучше других.

РЕШЕНИЕ

Все перечисленные классы коллекций предназначены для хранения ассоциативных пар «ключ/значение» и поддерживают средства перебора ключей. Самое важное различие — гарантии временной сложности и упорядочение ключей.

- `HashMap` обеспечивает поиск и вставку за время $O(1)$. При переборе ключей порядок их следования фактически непредсказуем. Коллекция реализуется в виде массива связных списков.
- `TreeMap` обеспечивает поиск и вставку за время $O(\log N)$. Ключи упорядочены, поэтому если их перебор должен осуществляться в порядке сортировки, такая возможность существует. Это означает, что ключи должны реализовать интерфейс `isComparable`. Коллекция `TreeMap` реализуется в виде красно-черного дерева.
- `LinkedHashMap` обеспечивает поиск и вставку за время $O(1)$. Ключи упорядочены в порядке вставки. Коллекция реализуется в виде двусвязных блоков.

Представьте, что пустые экземпляры `TreeMap`, `HashMap` и `LinkedHashMap` передаются следующей функции:

```
1  void insertAndPrint(AbstractMap<Integer, String> map) {
2    int[] array = {1, -1, 0};
3    for (int x : array) {
4      map.put(x, Integer.toString(x));
5    }
6
7    for (int k : map.keySet()) {
8      System.out.print(k + ", ");
9    }
10 }
```

Результаты для разных коллекций выглядят так:

HashMap	LinkedListMap	TreeMap
(произвольный порядок)	{1, -1, 0}	{-1, 0, 1}

Очень важно: результаты `LinkedListMap` и `TreeMap` должны выглядеть так, как показано выше. Для `HashMap` в моих тестах выводился результат `{0, 1, -1}`, но порядок может быть любым. Никаких гарантий на этот счет нет.

Когда упорядочение может понадобиться в реальной ситуации?

- Допустим, вы создаете отображение имен на объекты `Person`. Время от времени требуется выводить список людей, упорядоченный по имени в алфавитном порядке. `TreeMap` позволит вам это сделать.
- Класс `TreeMap` также предоставляет возможность вывести следующие 10 человек для заданного имени, например для реализации кнопки **Далее>** во многих приложениях.
- Класс `LinkedHashMap` удобен в тех случаях, когда порядок ключей должен соответствовать порядку вставки. В частности, данная возможность пригодится для организации кэширования, когда потребуется удалить самый старый элемент.

В общем случае рекомендуется использовать HashMap, если только у вас нет веских причин для другого выбора, а именно, если вам потребуется получить ключи в порядке вставки, используйте LinkedListMap. Если ключи должны возвращаться в фактическом/естественном порядке, используйте TreeMap. В остальных случаях класс HashMap, вероятно, подойдет лучше всего: обычно он работает быстрее и эффективнее.

13.6. Объясните, что такое рефлексия (reflection) объектов в Java и для чего она используется.

РЕШЕНИЕ

Рефлексия (или отражение) — механизм получения содержательной информации о классах и объектах Java, а также выполнение следующих операций:

1. Получение информации о методах и полях класса во время выполнения.
2. Создание нового экземпляра класса.
3. Прямое чтение и запись значений полей объекта по ссылке (независимо от модификатора доступа).

Пример использования рефлексии:

```
1  /* Параметры */
2  Object[] doubleArgs = new Object[] { 4.2, 3.9 };
3
4  /* Получение класса */
5  Class rectangleDefinition = Class.forName("MyProj.Rectangle");
6
7  /* Эквивалентно: Rectangle rectangle = new Rectangle(4.2, 3.9); */
8  Class[] doubleArgsClass = new Class[] {double.class, double.class};
9  Constructor doubleArgsConstructor =
10    rectangleDefinition.getConstructor(doubleArgsClass);
11  Rectangle rectangle = (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);
12
13  /* Эквивалентно: Double area = rectangle.area(); */
14  Method m = rectangleDefinition.getDeclaredMethod("area");
15  Double area = (Double) m.invoke(rectangle);
```

Этот код эквивалентен следующему:

```
1  Rectangle rectangle = new Rectangle(4.2, 3.9);
2  Double area = rectangle.area();
```

Зачем нужна рефлексия

Конечно, в приведенном выше примере рефлексия не кажется необходимой, но в некоторых случаях она очень полезна. Три основных причины для применения рефлексии:

1. Она упрощает наблюдение или управление поведением программы во время выполнения.
2. Она способствует отладке или тестированию программ, поскольку мы получаем прямой доступ к методам, конструкторам и полям.

3. Она позволяет вызывать методы по имени, даже если нам это имя заранее не известно. Например, пользователь может передать имя класса, параметры конструктора и имя метода, а программа использует эту информацию для создания объекта и вызова метода. Выполнение аналогичных операций без рефлексии потребует использования цепочки сложных команд `if` (если оно вообще возможно).

13.7. Существует класс `Country`, содержащий методы `getContinent()` и `getPopulation()`. Напишите функцию `int getPopulation(List<Country> countries, String continent)`, которая вычисляет население заданного континента по списку всех стран и названию континента с использованием лямбда-выражений.

РЕШЕНИЕ

Решение в действительности состоит из двух частей. Сначала нужно сгенерировать список стран (допустим, в Северной Америке), а затем вычислить их общее население.

Без лямбда-выражений это делается достаточно просто.

```
int getPopulation(List<Country> countries, String continent) {
   int sum = 0;
   for (Country c : countries) {
     if (c.getContinent().equals(continent)) {
       sum += c.getPopulation();
     }
   }
   return sum;
}
```

Чтобы написать реализацию с лямбда-выражениями, разобъем задачу на несколько подзадач.

Сначала мы используем фильтр для получения списка стран заданного континента.

```
Stream<Country> northAmerica = countries.stream().filter(
  country -> { return country.getContinent().equals(continent);}
);
```

Затем результат преобразуется в список с населениями стран:

```
Stream<Integer> populations = northAmerica.map(
  c -> c.getPopulation()
);
```

Наконец, общее население вычисляется методом `reduce`:

```
int population = populations.reduce(0, (a, b) -> a + b);
```

Следующая функция собирает все подзадачи воедино:

```
int getPopulation(List<Country> countries, String continent) {
  /* Фильтрация списка стран. */
  Stream<Country> sublist = countries.stream().filter(
    country -> { return country.getContinent().equals(continent);}
  );

```

```
7    /* Преобразование в список численности населения. */
8     Stream<Integer> populations = sublist.map(
9      c -> c.getPopulation()
10    );
11
12    /* Суммирование списка. */
13     int population = populations.reduce(0, (a, b) -> a + b);
14     return population;
15  }
```

Также из-за особенностей этой конкретной задачи можно обойтись вообще без фильтра. Операция `reduce` может содержать логику, которая связывает население стран других континентов с нулем. Таким образом, в сумме вклад других континентов учитываться не будет.

```
1  int getPopulation(List<Country> countries, String continent) {
2     Stream<Integer> populations = countries.stream().map(
3      c -> c.getContinent().equals(continent) ? c.getPopulation() : 0);
4     return populations.reduce(0, (a, b) -> a + b);
5  }
```

Лямбда-функции появились в Java 8, и если вы о них не знаете, удивляться не приходится. Зато у вас появляется хороший повод изучить их.

13.8. Напишите функцию `List<Integer> getRandomSubset(List<Integer> list)`, возвращающую случайное подмножество произвольного размера. Все подмножества (включая пустое) должны выбираться с одинаковой вероятностью. При написании функции следует использовать лямбда-выражения.

РЕШЕНИЕ

На первый взгляд можно пойти простым путем: выбрать размер подмножества от 0 до N, а затем сгенерировать случайное множество такого размера.

Однако такой подход порождает две проблемы:

1. Вероятностям придется назначать весовые коэффициенты. Если $N>1$, то количество подмножеств с размером $N/2$ больше количества подмножеств с размером N (которое, конечно, всегда равно 1).
2. Сгенерировать подмножество ограниченного размера (например, 10) сложнее, чем сгенерировать подмножество произвольного размера.

Вместо того чтобы генерировать подмножество на основании размера, рассмотрим задачу генерирования на основании элементов. (Тот факт, что в задаче предлагается использовать лямбда-выражения, также подсказывает, что искать нужно некоторую форму перебора с обработкой элементов.)

Представьте, что мы перебираем множество `{1, 2, 3}` для создания подмножества. Должен ли элемент 1 войти в подмножество?

Есть два варианта ответа: «да» и «нет». Вероятностям выбора следует присвоить весовые коэффициенты на основании доли подмножеств, включающих 1. Итак, какая часть подмножества содержит 1?

Для каждого отдельного элемента количество подмножеств, включающих его, равно количеству подмножеств, в которые этот элемент не входит. Рассмотрим следующий пример:

```
{}    {1}
{2}     {1, 2}
{3}     {1, 3}
{2, 3}   {1, 2, 3}
```

Обратите внимание: подмножества в левом и правом столбце отличаются только вхождением 1. Количество подмножеств в левом и правом столбце одинаково, потому что одно подмножество преобразуется в другое простым добавлением элемента.

Это означает, что для построения случайного подмножества достаточно перебрать список и «бросить монетку» (то есть принять решение с вероятностью 50/50), чтобы решить, должен ли каждый элемент войти в это множество.

Без лямбда-выражений реализация выглядит примерно так:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    List<Integer> subset = new ArrayList<Integer>();
3    Random random = new Random();
4    for (int item : list) {
5      /* Да/нет */
6      if (random.nextBoolean()) {
7        subset.add(item);
8     }
9    }
10    return subset;
11  }
```

Реализация с лямбда-выражениями:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    Random random = new Random();
3    List<Integer> subset = list.stream().filter(
4      k -> { return random.nextBoolean(); /* Да/нет */
5    }).collect(Collectors.toList());
6    return subset;
7  }
```

Также можно воспользоваться предикатом (определяемым в классе или функции):

```
1  Random random = new Random();
2  Predicate<Object> flipCoin = o -> {
3    return random.nextBoolean();
4  };
5
6  List<Integer> getRandomSubset(List<Integer> list) {
7    List<Integer> subset = list.stream().filter(flipCoin).
8      collect(Collectors.toList());
9    return subset;
10 }
```

Одно из преимуществ этой реализации — возможность применения предиката `flipCoin` в других местах.

14

Базы данных

В вопросах 1–3 используется следующая схема базы данных:

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

Обратите внимание, что у квартиры (Apartments) может быть несколько арендаторов (Tenants), а один арендатор может арендовать несколько квартир. Квартира может находиться только в одном здании (Buildings), а здание — в одном комплексе (Complexes).

14.1. Напишите SQL-запрос для получения списка арендаторов, которые снимают более одной квартиры.

РЕШЕНИЕ

Для реализации этого SQL-запроса можно воспользоваться условиями `HAVING` и `GROUP BY`, а затем выполнить операцию `INNER JOIN` с таблицей `Tenants`.

```
1  SELECT TenantName
2  FROM Tenants
3  INNER JOIN
4    (SELECT TenantID FROM AptTenants GROUP BY TenantID HAVING count(*) > 1) C
5  ON Tenants.TenantID = C.TenantID
```

Каждый раз при использовании на собеседовании (или на практике) `GROUP BY`, убедитесь, что в `SELECT` указана агрегатная функция или нечто, присутствующее в выражении `GROUP BY`.

14.2. Напишите SQL-запрос для получения списка всех зданий и количества открытых запросов (запросов со статусом `'open'`).

РЕШЕНИЕ

Эта задача использует прямое соединение таблиц `Requests` и `Apartments` для получения списка `ID` зданий и количества открытых запросов. Полученный список соединяется с таблицей `Buildings`.

```
1  SELECT BuildingName, ISNULL(Count, 0) as 'Count'
2  FROM Buildings
3  LEFT JOIN
4    (SELECT Apartments.BuildingID, count(*) as 'Count'
5     FROM Requests INNER JOIN Apartments
6     ON Requests.AptID = Apartments.AptID
7     WHERE Requests.Status = 'Open'
8     GROUP BY Apartments.BuildingID) ReqCounts
9  ON ReqCounts.BuildingID = Buildings.BuildingID
```

Подобные запросы с подзапросами необходимо тщательно протестировать. Вначале проверьте внутреннюю часть запроса, а потом — внешнюю.

14.3. Дом № 11 находится на капитальном ремонте. Напишите запрос, который закрывает все запросы на квартиры в этом здании.

РЕШЕНИЕ

Запросы `UPDATE`, как и запросы `SELECT`, могут содержать условие `WHERE`. Для реализации этого запроса нам понадобится список всех идентификаторов (`ID`) квартир в доме № 11, чтобы обновить их статус с помощью `UPDATE`.

```
1  UPDATE Requests
2  SET Status = 'Closed'
3  WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

14.4. Какие существуют типы соединений? Объясните, чем они различаются и почему определенные типы лучше подходят для конкретных ситуаций.

РЕШЕНИЕ

Ключевое слово `JOIN` выполняет соединение двух таблиц. Каждая из соединяемых таблиц должна иметь хотя бы одно поле, которое можно использовать для поиска соответствующих записей в другой таблице. Тип связи определяет, какие записи попадут в результирующий набор.

Возьмем две таблицы: в одной перечислены обычные напитки, а в другой — низкокалорийные. Каждая таблица состоит из двух полей: название напитка и код продукта. Поле `code` будет использоваться для поиска соответствия записей.

Обычные напитки (Beverage)

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA
Pepsi	PEPSI

Низкокалорийные напитки (Calorie-Free Beverages)

Name	Code
Diet Coca-Cola	COCACOLA
Fresca	FRESCA
Diet Pepsi	PEPSI
Pepsi Light	PEPSI
Purified Water	Water

Существует много способов связывания этих двух таблиц.

- `INNER JOIN`: результат будет содержать только данные, соответствующие указанному критерию. В нашем случае мы получим только три записи: одну с кодом `COCACOLA` и две с кодом `PEPSI`.
- `OUTER JOIN`: всегда содержит результаты `INNER JOIN`, но может содержать отдельные записи, не имеющие соответствий. Внешние объединения разделяются на следующие подтипы:
  - `LEFT OUTER JOIN` или `LEFT JOIN`: результат будет содержать все записи из левой таблицы. Если совпадений с правой таблицей нет, ее поля будут содержать значения `NULL`. В нашем примере мы получим 4 записи: в дополнение к `INNER JOIN` в результатах будет `BUDWEISER`, потому что он был в левой таблице.
  - `RIGHT OUTER JOIN` или `RIGHT JOIN` — противоположность `LEFT JOIN`. Результат будет содержать все записи из правой таблицы, а отсутствующие поля из левой таблицы будут содержать `NULL`. Обратите внимание: если для двух таблиц `A` и `B` выполняется соединение `A LEFT JOIN B`, то результат будет эквивалентен `B RIGHT JOIN A`. В нашем примере мы получим 5 записей, к результатам `INNER JOIN` добавятся `FRESCA` и `WATER`.
  - `FULL OUTER JOIN` (полное внешнее объединение) объединяет результаты `LEFT` и `RIGHT`. В результат попадут все записи из обеих таблиц, независимо от того, есть ли соответствие. При отсутствии соответствия поля принимают значение `NULL`. В нашем примере результат будет состоять из 6 записей.

14.5. Что такое денормализация? Поясните ее достоинства и недостатки.

РЕШЕНИЕ

Денормализация — метод оптимизации базы данных, при котором к одной или нескольким таблицам добавляются избыточные данные. Избыточность позволяет избежать высокозатратных соединений в реляционной базе данных.

В традиционной нормализованной базе данных данные хранятся в отдельных логических таблицах, а избыточность сводится к минимуму. Проектировщик стремится к тому, чтобы все фрагменты данных хранились только в одном экземпляре.

В нормализованной базе данных могут существовать таблицы `Courses` и `Teachers`. Каждая запись в `Courses` хранит `teacherID` для `Course`, но не `teacherName`. Когда потребуется получить список всех курсов с именами преподавателей, необходимо будет соединить эти две таблицы.

С одной стороны, это хорошо: если имя учителя изменится, информацию достаточно обновить только в одном месте.

С другой стороны, если таблицы будут большими, нам придется потратить слишком много времени на соединение таблиц.

Денормализация — это компромисс. При денормализации проектировщик сознательно решает, что некоторые данные должны храниться с избыточностью, но эффективность работы повышается за счет уменьшения количества соединений.

Недостатки денормализации	Преимущества денормализации
Операции обновления и вставки записей весьма затратны	Ускорение выборки данных (из-за сокращения количества соединений)
Усложнение запросов на обновление и вставку записей	Упрощение запросов (а следовательно, снижение вероятности ошибки), так как в выборке задействовано меньше таблиц
Появляется возможность нарушения целостности данных. Какое из различающихся значений следует считать «правильным»?	
Избыточность данных повышает затраты пространства	

В системах с особыми требованиями к масштабированию (как в крупнейших компаниях) всегда используются и нормализованные, и денормализованные базы данных.

14.6. Нарисуйте диаграмму отношений для базы данных, в которой фигурируют компании, клиенты и сотрудники компаний.

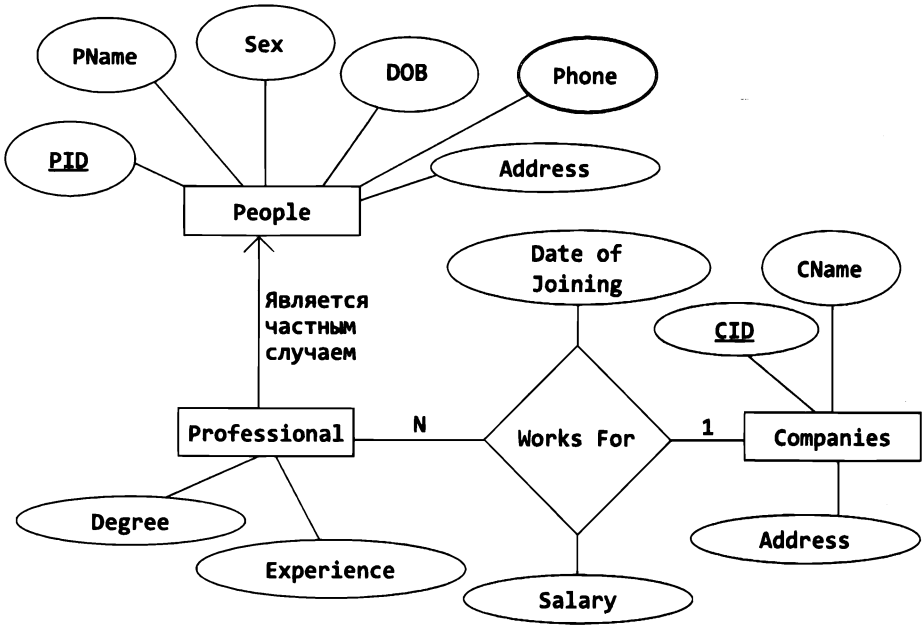
РЕШЕНИЕ

Сотрудники, работающие в компаниях (`Company`), являются профессионалами (`Professional`). Таким образом, между `Professional` и `People` (человек) существует отношение типа «является частным случаем» (также можно сказать, что `Professional` является производным от `People`).

Кроме свойств, полученных от `People`, у каждого объекта `Professional` существует дополнительная информация — например, ученая степень и стаж работы.

Профессионал работает только на одну компанию, но компания может нанять много профессионалов. Таким образом, между `Professional` и `Company` существует связь «много-к-одному». Это отношение (`Works For`) может содержать дополнительные атрибуты (например, дату начала работы и зарплату). Эти атрибуты определяются при создании связи `Professional` с `Company`.

У человека (`Person`) может быть несколько номеров телефона, поэтому атрибут `Phone` может иметь множественные значения.



14.7. Разработайте простую базу данных, содержащую данные об успеваемости студентов, и напишите запрос, возвращающий список лучших студентов (лучшие 10 %), отсортированный по их среднему баллу.

РЕШЕНИЕ

Простая база данных будет содержать не меньше трех таблиц: `Students` (Студенты), `Courses` (Курс лекций) и `CourseEnrollment` (Посещение курса). В `Students` хранится как минимум имя студента и его идентификатор, а также, вероятно, некоторые персональные данные. В `Courses` хранится название курса и идентификатор, а также, вероятно, дополнительная информация: описание курса, имя преподавателя и т. д. `CourseEnrollment` объединяет `Students` и `Courses`, а также содержит поле `CourseGrade` (итоговая оценка).

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	int
Term	int

При желании эту базу данных можно произвольно усложнить, добавив в нее информацию о преподавателе, зарплате и другие данные.

Первая (и неправильная) версия запроса, использующая функцию Microsoft SQL Server `TOP ... PERCENT`, может выглядеть так:

```
SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
                      CourseEnrollment.StudentID
FROM CourseEnrollment
GROUP BY CourseEnrollment.StudentID
ORDER BY AVG(CourseEnrollment.Grade)
```

Проблема в том, что этот код буквально возвращает верхние 10% записей, отсортированных по среднему баллу.

Представьте, что имеется выборка из 100 студентов, и 15 лучших студентов имеют одинаковый средний балл. Приведенная выше функция вернет только 10 из этих студентов — не то, что требуется. При равенстве среднего балла в результат должны быть включены все студенты, даже если их численность превысит 10% группы.

Для решения этой проблемы можно использовать аналогичный запрос, но предварительно вычислить пороговый средний балл:

```
DECLARE @GPACutOff float;
SET @GPACutOff = (SELECT min(GPA) as 'GPAMin' FROM (
    SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
    FROM CourseEnrollment
    GROUP BY CourseEnrollment.StudentID
    ORDER BY GPA desc) Grades);
```

После того как значение `@GPACutOff` будет определено, выборка студентов со средним баллом не ниже вычисленного выполняется относительно просто.

```
SELECT StudentName, GPA
FROM (SELECT AVG(CourseEnrollment.Grade) AS GPA, CourseEnrollment.StudentID
      FROM CourseEnrollment
      GROUP BY CourseEnrollment.StudentID
      HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

Будьте очень осторожны с неявными предположениями. Взгляните еще раз на описание базы данных. Какое потенциально ошибочное предположение вы видите? То, что один курс может читаться только одним преподавателем. А ведь в некоторых случаях один и тот же курс могут вести несколько преподавателей.

Впрочем, совсем без предположений все равно не обойтись. Более того, сами предполжения менее важны, чем осознание того факта, что вы *сделали* предположения. Даже неправильные предположения — как в реальной работе, так и в собеседовании — допустимы, *если вы признаете их существование.*

Помните, что вам нужно найти компромисс между гибкостью и сложностью. Создание системы, в которой курс могут читать несколько преподавателей, повышает гибкость базы данных, но и увеличивает ее сложность. Если попытаться сделать БД, учитывающую все возможные ситуации, то результат может оказаться слишком сложным.

Сделайте свой проект в меру гибким и явно изложите любые предположения или ограничения. Это касается не только проектирования баз данных, но и объектно-ориентированного проектирования и программирования в целом.

15

Потоки и блокировки

15.1. В чем разница между потоком и процессом?

РЕШЕНИЕ

Процессы и потоки — взаимосвязанные понятия, но между ними также существуют принципиальные различия.

Процесс может рассматриваться как экземпляр выполняемой программы. По сути это независимый объект, которому выделяются системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс захочет получить доступ к чужим ресурсам, необходимо использовать средства межпроцессного взаимодействия: конвейеры, файлы, сокеты и многое другое.

Поток существует внутри процесса и использует его ресурсы (включая пространство кучи). Множественные потоки внутри одного процесса совместно работают с одной кучей — в отличие от процессов, которые не могут просто так обратиться к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут выполнять операции чтения и записи с памятью кучи.

Поток представляет ветвь выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видимым другим потокам этого процесса.

15.2. Как оценить время, затрачиваемое на переключение контекста?

РЕШЕНИЕ

Вопрос непростой, но давайте рассмотрим одно из возможных решений.

Переключение контекста — это время, потраченное на переключение между двумя процессами (то есть перевод ожидающего процесса в состояние выполнения и перевод текущего выполняемого процесса в состояние ожидания/завершения). Такие ситуации возникают в многозадачных системах. Операционная система загружает информацию о состоянии ожидающих процессов в память и сохраняет информацию о состоянии работающего процесса.

Для решения этой задачи понадобится зарегистрировать временные метки для последней и первой инструкций переключаемых процессов. Время переключения контекста равно разности между временными метками двух процессов.

Рассмотрим простой пример. Предположим, что существует всего два процесса: P_1 и P_2. Процесс P_1 выполняется, а P_2 ожидает своей очереди. В некоторый момент времени операционная система должна поменять местами P_1 и P_2. Предположим, что замена происходит на N-й инструкции процесса P_1. Если $t_{x,k}$ — временная метка (в микросекундах) k-й инструкции процесса x, то переключение контекста займет $t_{2,1} - t_{1,n}$ микросекунд.

Теперь самое сложное: как узнать, что произошло переключение процесса? Конечно же, мы не сможем сохранять временную метку для каждой инструкции процесса.

Другая проблема заключается в том, что переключением процессов управляет алгоритм планирования операционной системы, а в системе могут существовать многочисленные потоки уровня ядра, которые также выполняют переключения контекста. За процессор или прерывания системных вызовов ядра может конкурировать множество процессов. У пользователя нет возможности контролировать эти «посторонние» контекстные переключения. Например, если в момент времени $t_{1,n}$ ядро решит обработать прерывание, то время переключения контекста окажется завышенным.

Чтобы преодолеть эти препятствия, нужно разработать такую среду, чтобы после запуска P_1 планировщик задач сразу же выбрал процесс P_2. Это может быть реализовано с помощью канала передачи данных между P_1 и P_2: наличие двух процессов, которые будут передавать туда-сюда маркер данных.

Пусть P_1 будет инициатором: он будет отправлять данные, а P_2 — получать. P_2 блокируется («спит») в ожидании данных от P_1. При запуске P_1 передает по каналу данных маркер (токен) процессу P_2 и сразу пытается прочитать ответ. Поскольку P_2 еще не начал работать, информации для отправки процессу P_1 не существует; процесс блокируется, а процессор освобождается.

Происходит переключение контекста, и планировщик задач должен выбрать другой процесс. Так как P_2 подготовлен к работе, он автоматически будет выбран планировщиком для выполнения. Во время выполнения P_2 роли P_1 и P_2 меняются. Теперь P_2 работает как отправитель, а P_1 — заблокированный получатель. Игра заканчивается, когда P_2 возвращает маркер P_1.

Подведем итог. Последовательность шагов выглядит так:

1. P_2 блокируется в ожидании данных от P_1.
2. P_1 отмечает время начала.
3. P_1 отправляет маркер процессу P_2.
4. P_1 пытается прочитать ответ от P_2. Это приводит к переключению контекста.
5. P_2 активизируется планировщиком и получает маркер.
6. P_2 отправляет ответный маркер процессу P_1.
7. P_2 пытается прочитать ответный маркер от P_1. Это приводит к переключению контекста.
8. P_1 актизивируется планировщиком и получает маркер.
9. P_1 отмечает время завершения.

Ключ к решению этой задачи в том, что отправка маркера данных порождает переключение контекста. Допустим, T_d и T_r — время отправки и получения маркера

соответственно, а T_c — общее время, затраченное на контекстное переключение. На втором шаге P_1 записывает время отправки маркера, а на девятом — записывает время получения. Общее время T (между двумя событиями) выражается так:

$$T = 2 \times (T_d + T_c + T_r)$$

Эта формула легко выводится из следующей последовательности событий: P_1 отправляет маркер (3), процессор выполняет переключение контекста (4), P_2 получает маркер (5). Затем P_2 отправляет ответный маркер (6), процессор опять выполняет переключение контекста (7) и, наконец, P_1 получает маркер обратно (8).

Процесс P_1 может рассчитать T, поскольку это промежуток времени между событиями 3 и 8. Так, чтобы вычислить T_c, нам нужно сначала получить значение $T_d + T_r$.

Как узнать эту информацию? Можно измерить отрезок времени, который P_1 тратит на отправку и получение маркера с самим собой. Это не приведет к переключению контекста, так как P_1 выполняется в момент отправки и не блокируется для получения маркера.

Процедура многократно повторяется для исключения отклонений во времени выполнения шагов 2–9, которые могут быть обусловлены неожиданными прерываниями ядра или дополнительными потоками, конкурирующими за процессорное время. В качестве окончательного ответа выбирается минимальное зарегистрированное значение времени.

Тем не менее в конечном итоге можно утверждать лишь то, что полученный результат — приближенное значение, зависящее от конкретной системы. Например, мы предполагаем, что поток P_2 будет выбран для запуска при поступлении маркера данных. Однако все зависит от планировщика задач, и никаких гарантий на этот счет дать нельзя.

И это нормально. Главное, не забудьте на собеседовании отметить, что ваше решение может оказаться не идеальным.

15.3. В знаменитой задаче об обедающих философах каждый из них имеет только одну палочку для еды. Философу нужно две палочки, и он всегда поднимает левую палочку, а потом — правую. Взаимная блокировка произойдет, если все философы будут одновременно использовать левую палочку. Используя потоки и блокировки, реализуйте модель задачи, предотвращающую взаимные блокировки.

РЕШЕНИЕ

Для начала реализуем упрощенную модель обеденной задачи, не учитывающую возможность взаимных блокировок. Нам понадобятся классы `Philosopher` (философ), расширяющий `Thread`, и `Chopstick` (палочка для еды): когда пользователь берет ее, вызывается метод `lock.lock()`, а когда кладет — метод `lock.unlock()`.

```
1  class Chopstick {
2    private Lock lock;
3
4    public Chopstick() {
5      lock = new ReentrantLock();
6    }
7
```

```
8    public void pickUp() {
9      void lock.lock();
10   }
11
12   public void putDown() {
13     lock.unlock();
14   }
15 }
16
17 class Philosopher extends Thread {
18   private int bites = 10;
19   private Chopstick left, right;
20
21   public Philosopher(Chopstick left, Chopstick right) {
22     this.left = left;
23     this.right = right;
24   }
25
26   public void eat() {
27     pickUp();
28     chew();
29     putDown();
30   }
31
32   public void pickUp() {
33     left.pickUp();
34     right.pickUp();
35   }
36
37   public void chew() { }
38
39   public void putDown() {
40     right.putDown();
41     left.putDown();
42   }
43
44   public void run() {
45     for (int i = 0; i < bites; i++) {
46       eat();
47    }
48   }
49 }
```

Выполнение этого кода может привести к взаимной блокировке, если все философы возьмут левую палочку и будут ждать правую.

Решение 1. Все или ничего

Чтобы предотвратить взаимную блокировку, можно реализовать следующую стратегию: если философ не может взять правую палочку, он кладет левую на место.

```
1  public class Chopstick {
2    /* См. выше */
3
4    public boolean pickUp() {
5      return lock.tryLock();
6    }
```

```
}

public class Philosopher extends Thread {
  /* См. выше */

  public void eat() {
    if (pickUp()) {
      chew();
      putDown();
    }
  }

  public boolean pickUp() {
    /* Попытка взять палочку */
    if (!left.pickUp()) {
      return false;
    }
    if (!right.pickUp()) {
      left.putDown();
      return false;
    }
    return true;
  }
}
```

В этом коде необходимо следить за тем, чтобы философ клал на место левую палочку, если он не смог взять правую, и не вызывал метод `putDown()`, если он не держит палочку.

Решение 2. Назначение приоритетов

Также можно пометить палочки номерами от 0 до `N-1`. Каждый философ сначала пытается взять палочку с меньшим номером. Фактически это означает, что каждый философ берет левую палочку перед правой (если номера были назначены именно так), кроме последнего философа, который поступает наоборот. Тем самым цикл разрывается.

```
public class Philosopher extends Thread {
  private int bites = 10;
  private Chopstick lower, higher;
  private int index;
  public Philosopher(int i, Chopstick left, Chopstick right) {
    index = i;
    if (left.getNumber() < right.getNumber()) {
      this.lower = left;
      this.higher = right;
    } else {
      this.lower = right;
      this.higher = left;
    }
  }

  public void eat() {
    pickUp();
    chew();
    putDown();
```

```
20    }
21
22    public void pickUp() {
23      lower.pickUp();
24      higher.pickUp();
25    }
26
27    public void chew() { ... }
28
29    public void putDown() {
30      higher.putDown();
31      lower.putDown();
32    }
33
34    public void run() {
35      for (int i = 0; i < bites; i++) {
36        eat();
37     }
38    }
39  }
40
41  public class Chopstick {
42    private Lock lock;
43    private int number;
44
45    public Chopstick(int n) {
46      lock = new ReentrantLock();
47      this.number = n;
48    }
49
50    public void pickUp() {
51      lock.lock();
52    }
53
54    public void putDown() {
55      lock.unlock();
56    }
57
58    public int getNumber() {
59      return number;
60    }
61  }
```

При таком решении философ никогда не держит «бóльшую» палочку, если он при этом не держит «меньшую». Тем самым предотвращается возможность циклической блокировки.

15.4. Разработайте класс, предоставляющий блокировку только в том случае, если это не приведет к созданию взаимной блокировки.

РЕШЕНИЕ

Существует несколько общих способов предотвращения взаимных блокировок. Одно из самых популярных решений — заставить процесс явно объявлять, какие блокировки ему нужны. Тогда программа может проверить, не приведет ли

получение такой блокировки к созданию взаимной блокировки, и если приведет — вернуть признак ошибки.

Как обнаружить взаимную блокировку? Предположим, что блокировки запрашиваются в следующем порядке:

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

Это приведет к взаимной блокировке, потому что:

```
A блокирует 2, ждет 3
B блокирует 3, ждет 5
C блокирует 5, ждет 2
```

Можно представить эту ситуацию в виде графа, где узел 2 соединен с 3, узел 3 соединен с 5, а узел 5 соединен с 2. Взаимная блокировка представляется циклом в графе. Ребро (`w`, `v`) существует в графе, если процесс объявляет, что он запрашивает блокировку `v` немедленно после блокировки `w`. В предыдущем примере в графе будут существовать следующие ребра: (`1, 2`), (`2, 3`), (`3, 4`), (`1, 3`), (`3, 5`), (`7, 5`), (`5, 9`), (`9, 2`). «Владелец» ребра не имеет значения.

Этому классу потребуется метод `declare`, который использует потоки и процессы для объявления порядка запроса ресурсов. Метод `declare` перебирает порядок объявления, добавляя каждую непрерывную пару элементов (`v`, `w`) в граф. Далее он проверяет, не присутствует ли в графе цикл. При обнаружении цикла добавленное ребро удаляется из графика, после чего метод возвращает управление.

Остается решить одну проблему. Как обнаружить цикл? Мы можем обнаружить цикл с помощью поиска в глубину через каждый связанный элемент (то есть через каждый компонент графа). Существуют сложные алгоритмы для поиска всех связных компонент графа, но наша задача не требует такой сложности.

Мы знаем, что если возникает цикл, то виновато одно из ребер. Если поиск в глубину в какой-то момент посещает каждое ребро, мы знаем, что поиск цикла завершен.

Псевдокод упрощенного обнаружения циклов выглядит примерно так:

```
1  boolean checkForCycle(locks[] locks) {
2    touchedNodes = hash table(lock -> boolean)
3    Инициализировать touchedNodes значением false для всех блокировок из locks
4    for each (lock x in process.locks) {
5      if (touchedNodes[x] == false) {
6        if (hasCycle(x, touchedNodes)) {
7          return true;
8      }
9     }
10    }
11    return false;
12 }
13
14 boolean hasCycle(node x, touchedNodes) {
15    touchedNodes[r] = true;
16    if (x.state == VISITING) {
17      return true;
18    } else if (x.state == FRESH) {
```

```
19      ... (См. ниже полный код)
20    }
21  }
```

В этом коде могут выполняться несколько поисков в глубину, но `touchedNodes` инициализируется только один раз. Перебор ведется до тех пор, пока все значения в `touchedNodes` не примут значение `false`.

Далее приводится более подробная версия кода. Для простоты предполагается, что все блокировки и процессы (владельцы) последовательно упорядочены.

```
1  class LockFactory {
2    private static LockFactory instance;
3
4    private int numberOfLocks = 5; /* по умолчанию */
5    private LockNode[] locks;
6
7    /* Связывает процесс или владельца с порядком, в котором
8     * (по утверждению владельца) будут получаться блокировки */
9    private HashMap<Integer, LinkedList<LockNode>> lockOrder;
10
11    private LockFactory(int count) { ... }
12    public static LockFactory getInstance() { return instance; }
13
14    public static synchronized LockFactory initialize(int count) {
15      if (instance == null) instance = new LockFactory(count);
16      return instance;
17    }
18
19    public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes,
20                            int[] resourcesInOrder) {
21      /* Проверка цикла */
22      for (int resource : resourcesInOrder) {
23        if (touchedNodes.get(resource) == false) {
24          LockNode n = locks[resource];
25       if (n.hasCycle(touchedNodes)) {
26            return true;
27       }
28      }
29     }
30      return false;
31    }
32
33    /* Для предотвращения взаимных блокировок процессы должны заранее
34     * объявить свой порядок получения блокировок. Убедиться в том,
35     * что этот порядок не создает цикла в направленном графе. */
36    public boolean declare(int ownerId, int[] resourcesInOrder) {
37      HashMap<Integer, Boolean> touchedNodes = new HashMap<Integer, Boolean>();
38
39      /* Добавление узлов в граф */
40      int index = 1;
41      touchedNodes.put(resourcesInOrder[0], false);
42      for (index = 1; index < resourcesInOrder.length; index++) {
43        LockNode prev = locks[resourcesInOrder[index - 1]];
44        LockNode curr = locks[resourcesInOrder[index]];
45        prev.joinTo(curr);
46        touchedNodes.put(resourcesInOrder[index], false);
```

```
47      }
48
49      /* Если возникает цикл, уничтожить список ресурсов и вернуть false */
50      if (hasCycle(touchedNodes, resourcesInOrder)) {
51        for (int j = 1; j < resourcesInOrder.length; j++) {
52          LockNode p = locks[resourcesInOrder[j - 1]];
53          LockNode c = locks[resourcesInOrder[j]];
54          p.remove(c);
55      }
56        return false;
57      }
58
59      /* Циклы не обнаружены. Сохранить порядок объявления, чтобы
60       * убедиться в том, что процесс действительно получает блокировки
61       * в заявленном им порядке. */
62      LinkedList<LockNode> list = new LinkedList<LockNode>();
63      for (int i = 0; i < resourcesInOrder.length; i++) {
64        LockNode resource = locks[resourcesInOrder[i]];
65        list.add(resource);
66      }
67      lockOrder.put(ownerId, list);
68
69      return true;
70    }
71
72    /* Получить блокировку, предварительно убедившись в том, что процесс
73     * действительно получает блокировки в заявленном порядке. */
74    public Lock getLock(int ownerId, int resourceID) {
75      LinkedList<LockNode> list = lockOrder.get(ownerId);
76      if (list == null) return null;
77
78      LockNode head = list.getFirst();
79      if (head.getId() == resourceID) {
80        list.removeFirst();
81        return head.getLock();
82      }
83      return null;
84    }
85  }
86
87  public class LockNode {
88    public enum VisitState { FRESH, VISITING, VISITED  };
89
90    private ArrayList<LockNode> children;
91    private int lockId;
92    private Lock lock;
93    private int maxLocks;
94
95    public LockNode(int id, int max) { ...   }
96
97    /* Соединить "this" с "node", убедившись в отсутствии циклов */
98    public void joinTo(LockNode node) { children.add(node); }
99    public void remove(LockNode node) { children.remove(node); }
100
101    /* Проверка цикла начинается с поиска в глубину. */
102    public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes) {
103      VisitState[] visited = new VisitState[maxLocks];
104      for (int i = 0; i < maxLocks; i++) {
```

```
105         visited[i] = VisitState.FRESH;
106       }
107       return hasCycle(visited, touchedNodes);
108     }
109
110     private boolean hasCycle(VisitState[] visited,
111                              HashMap<Integer, Boolean> touchedNodes) {
112       if (touchedNodes.containsKey(lockId)) {
113         touchedNodes.put(lockId, true);
114       }
115
116       if (visited[lockId] == VisitState.VISITING) {
117         /* Возврат к уже посещенному узлу - обнаружен цикл. */
119         return true;
120       } else if (visited[lockId] == VisitState.FRESH) {
121         visited[lockId] = VisitState.VISITING;
122         for (LockNode n : children) {
123           if (n.hasCycle(visited, touchedNodes)) {
124             return true;
125         }
126       }
127         visited[lockId] = VisitState.VISITED;
128       }
129       return false;
130     }
131
132     public Lock getLock() {
133       if (lock == null) lock = new ReentrantLock();
134       return lock;
135     }
136
137     public int getId() { return lockId; }
138   }
```

Как обычно, никто не ждет, что вы напишете такой длинный и сложный код на собеседовании. Вероятно, вас попросят написать заготовку псевдокода и реализовать один из методов.

15.5. Имеется следующий код:

```
public class Foo {
  public Foo() {  ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

Один и тот же экземпляр `Foo` передается трем различным потокам: `ThreadA` вызывает `first`, `ThreadB` вызывает `second`, а `threadC` вызывает `third`. Разработайте механизм, гарантирующий, что метод `first` будет вызван перед `second`, а метод `second` будет вызван перед `third`.

РЕШЕНИЕ

Общая логика проста: перед вызовом `second()` нужно проверить, что вызов `first()` завершен, а перед вызовом `third()` — что завершен вызов `second()`. Мы должны крайне осторожно отнестись к потоковой безопасности кода, поэтому простых флагов будет недостаточно.

Нельзя ли воспользоваться блокировкой?

```
1  public class FooBad {
2    public int pauseTime = 1000;
3    public ReentrantLock lock1, lock2;
4
5    public FooBad() {
6      try {
7        lock1 = new ReentrantLock();
8        lock2 = new ReentrantLock();
9
10       lock1.lock();
11       lock2.lock();
12     } catch (...) { ... }
13   }
14
15   public void first() {
16     try {
17     ...
18       lock1.unlock(); // Установить признак завершения first()
19     } catch (...) { ... }
20   }
21
22   public void second() {
23     try {
24       lock1.lock(); // Ожидать завершения first()
25       lock1.unlock();
26     ...
27
28       lock2.unlock(); // Установить признак завершения second()
29     } catch (...) { ... }
30   }
31
32   public void third() {
33     try {
34       lock2.lock(); // Ожидать завершения second()
35       lock2.unlock();
36     ...
37     } catch (...) { ... }
38   }
39 }
```

Однако этот код будет работать не так, как ожидается, из-за проблемы с *владельцем блокировки*. Один поток получает блокировку (конструктор `FooBad`), а другие потоки пытаются эту блокировку снять. Так поступать нельзя, в результате будет выдано исключение. Владельцем блокировки в Java является тот поток, который ее захватил.

Вместо этого мы можем использовать семафоры, логика работы останется той же:

```
1  public class Foo {
2    public Semaphore sem1, sem2;
3
4    public Foo() {
5      try {
6        sem1 = new Semaphore(1);
7        sem2 = new Semaphore(1);
8
9        sem1.acquire();
10       sem2.acquire();
11     } catch (...) { ... }
12   }
13
14   public void first() {
15     try {
16     ...
17       sem1.release();
18     } catch (...) { ... }
19   }
20
21   public void second() {
22     try {
23       sem1.acquire();
24       sem1.release();
25     ...
26       sem2.release();
27     } catch (...) { ... }
28   }
29
30   public void third() {
31     try {
32       sem2.acquire();
33       sem2.release();
34     ...
35     } catch (...) { ... }
36   }
37 }
```

15.6. Имеется класс с синхронизированным методом A и обычным методом B. Если у вас есть два потока в одном экземпляре программы, смогут ли они оба выполнить A одновременно? Могут ли они выполнять A и B одновременно?

РЕШЕНИЕ

Объявление метода с ключевым словом `synchronized` гарантирует, что два потока не могут одновременно выполнять этот метод *для одного экземпляра* объекта.

Таким образом, ответ на первую часть вопроса зависит от обстоятельств. Если два потока работают с одним экземпляром объекта, то они не могут одновременно выполнить метод `A`, а если экземпляры объекта разные, то могут.

В этом можно убедиться на примере блокировок. Синхронизируемый метод устанавливает «блокировку» для *всех* синхронизируемых методов в конкретном экземпляре объекта, запрещая любым потокам выполнять синхронизируемые методы этого объекта.

Во второй части вопроса речь идет о том, может ли поток `thread1` выполнять синхронизируемый метод `A`, в то время как поток `thread2` выполняет несинхронизируемый метод `B`. Так как метод `B` не синхронизирован, ничто не мешает `thread1` выполнять `A` в то время, как `thread2` выполняет `B`. Это утверждение остается истинным независимо от того, работают ли `thread1` и `thread2` с одним экземпляром объекта.

В конечном итоге нужно запомнить, что для каждого экземпляра объекта в любой момент времени может выполняться только один синхронизируемый метод. Другие потоки могут выполнять как несинхронизируемые методы для этого экземпляра, так и произвольные методы для другого экземпляра объекта.

15.7. В классической задаче требуется вывести последовательность чисел от 1 до n. Если число кратно 3, то выводится сообщение «Fizz». Если число кратно 5, то выводится сообщение «Buzz». Если число кратно и 3, и 5, выводится сообщение «FizzBuzz». Разработайте многопоточную реализацию этой задачи. Один поток проверяет кратность 3 и выводит «Fizz». Другой поток отвечает за проверку кратности 5 и выводит «Buzz». Третий поток отвечает за проверку кратности 3 и 5 и выводит «FizzBuzz». Четвертый поток работает с числами.

РЕШЕНИЕ

Начнем с реализации однопоточной версии задачи.

Однопоточная версия

Хотя эта задача (в однопоточном варианте) не особенно сложна, многие кандидаты излишне усложняют ее. Они ищут «красивое» решение, которое бы использовало тот факт, что случай кратности 3 и 5 («FizzBuzz») напоминает отдельные случаи («Fizz» и «Buzz»).

На самом деле лучшее решение (в отношении удобочитаемости и эффективности кода) оказывается одновременно и самым прямолинейным.

```
1  void fizzbuzz(int n) {
2    for (int i = 1; i <= n; i++) {
3      if (i % 3 == 0 && i % 5 == 0) {
4        System.out.println("FizzBuzz");
5      } else if (i % 3 == 0) {
6        System.out.println("Fizz");
7      } else if (i % 5 == 0) {
8        System.out.println("Buzz");
9      } else {
10       System.out.println(i);
11     }
12   }
13 }
```

Прежде всего обратите внимание на порядок команд. Если разместить проверку делимости на 3 перед проверкой делимости на 3 и 5, результат будет неправильным.

Многопоточная версия

Структура многопоточной реализации выглядит примерно так:

Поток FizzBuzz	Поток Fizz
если i делится на 3 и 5 вывести FizzBuzz увеличить i повторять до i > n	если i делится только на 3 вывести Fizz увеличить i повторять до i > n

Поток Buzz	Поток вывода чисел
если i делится только на 5 вывести Buzz увеличить i повторять до i > n	если i не делится ни на 3, ни на 5 вывести i увеличить i повторять до i > n

Код выглядит примерно так:

```
1  while (true) {
2    if (current > max) {
3      return;
4    }
5    if (/* Проверка делимости */) {
6      System.out.println(/* Вывод */);
7      current++;
8    }
9  }
```

В цикл следует добавить средства синхронизации. В противном случае значение `current` может измениться между строками 2–4 и 5–8, что может привести к непреднамеренному выходу за границы цикла. Кроме того, операция инкремента не обладает потоковой безопасностью.

Существует много возможностей фактической реализации этой концепции. Например, можно создать четыре разных класса потоков, совместно использующих одну ссылку на переменную `current` (которая может быть упакована в объект).

Циклы потоков во многом похожи: просто они используют разные значения для проверки кратности и выводят разные значения.

	FizzBuzz	Fizz	Buzz	Число
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false

В основном задача решается передачей параметра для проверки и выводимого значения. Впрочем, вывод потока `Number` необходимо заменить, потому что он не является простой фиксированной строкой.

Приведенная ниже реализация `FizzBuzzThread` решает большинство этих проблем. Класс `NumberThread` может расширить `FizzBuzzThread` и переопределить метод `print`.

```
Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
                    new FizzBuzzThread(true, false, n, "Fizz"),
                    new FizzBuzzThread(false, true, n, "Buzz"),
                    new NumberThread(false, false, n)};
for (Thread thread : threads) {
  thread.start();
}

public class FizzBuzzThread extends Thread {
   private static Object lock = new Object();
   protected static int current = 1;
   private int max;
   private boolean div3, div5;
   private String toPrint;

   public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
     this.div3 = div3;
     this.div5 = div5;
     this.max = max;
     this.toPrint = toPrint;
   }

   public void print() {
     System.out.println(toPrint);
   }

   public void run() {
     while (true) {
       synchronized (lock) {
         if (current > max) {
           return;
       }

         if ((current % 3 == 0) == div3 &&
             (current % 5 == 0) == div5) {
           print();
           current++;
       }
      }
     }
   }
}

public class NumberThread extends FizzBuzzThread {
   public NumberThread(boolean div3, boolean div5, int max) {
     super(div3, div5, max, null);
   }

   public void print() {
     System.out.println(current);
   }
}
```

Заметьте, что сравнение `current` и `max` должно предшествовать команде `if`; это гарантирует, что значение будет выводиться только в том случае, если `current` меньше либо равно `max`.

Также возможно другое решение: передать методы `validate` и `print` в параметрах (если ваш язык поддерживает такую возможность, как, например, Java 8 и многие другие).

```
int n = 100;
Thread[] threads = {
  new FBThread(i -> i % 3 == 0 && i % 5 == 0, i -> "FizzBuzz", n),
  new FBThread(i -> i % 3 == 0 && i % 5 != 0, i -> "Fizz", n),
  new FBThread(i -> i % 3 != 0 && i % 5 == 0, i -> "Buzz", n),
  new FBThread(i -> i % 3 != 0 && i % 5 != 0, i -> Integer.toString(i), n)};
for (Thread thread : threads) {
  thread.start();
}

public class FBThread extends Thread {
  private static Object lock = new Object();
  protected static int current = 1;
  private int max;
  private Predicate<Integer> validate;
  private Function<Integer, String> printer;
  int x = 1;

  public FBThread(Predicate<Integer> validate,
                  Function<Integer, String> printer, int max) {
    this.validate = validate;
    this.printer = printer;
    this.max = max;
  }

  public void run() {
    while (true) {
      synchronized (lock) {
        if (current > max) {
          return;
      }
      if (validate.test(current)) {
          System.out.println(printer.apply(current));
          current++;
      }
     }
    }
  }
}
```

Конечно, возможны и другие варианты реализации.

16

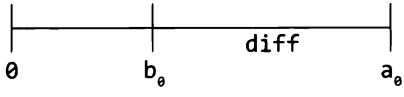
Задачи умеренной сложности

16.1. Напишите функцию, которая переставляет значения переменных «на месте» (то есть без использования временных переменных).

РЕШЕНИЕ

Это классическая задача, которую часто предлагают на собеседованиях, и она достаточно проста. Пусть a_0 — это исходное значение a, а b_0 — исходное значение b. Обозначим diff разницу $a_0 - b_0$.

Ниже изображено взаимное расположение всех этих значений на числовой оси для случая a > b:



Присвоим a значение diff. Если сложить значение b и diff, то мы получим a_0 (результат следует сохранить в b). Теперь $b = a_0$ и a = diff. Далее остается лишь присвоить b значение a_0 - diff, которое равно b - a.

Реализация этого алгоритма выглядит так:

```
1  // Пример для a = 9, b = 4
2  a = a - b; // a = 9 - 4 = 5
3  b = a + b; // b = 5 + 4 = 9
4  a = b - a; // a = 9 - 5
```

Аналогичное решение можно реализовать при помощи поразрядных операций. К преимуществам такого решения относится то, что оно позволяет работать с другими типами данных, кроме целых чисел.

```
1  // Пример для a = 101 и b = 110 (в двоичной записи)
2  a = a^b; // a = 101^110 = 011
3  b = a^b; // b = 011^110 = 101
4  a = a^b; // a = 011^101 = 110
```

Работа этого кода основана на операции XOR. Принцип работы этого кода проще всего понять на примере конкретного бита. Если нам удастся поменять местами два бита, значит, алгоритм работает правильно.

Рассмотрим работу алгоритма по шагам:

1. x = x ^ y

   Эта строка фактически проверяет, имеют ли x и y разные значения. Она возвращает 1 в том и только в том случае, если x != y.

2. `y = x ^ y`

   Или: `y = {0` *если исходные значения совпадают,* `1` *если различаются}* `^ {`*исходное* `y}`

   Заметим, что операция XOR с 1 всегда приводит к изменению состояния бита, а операция XOR с 0 никогда не изменяет его.

   Следовательно, при выполнении операции `y = 1 ^ {`*исходное* `y}` при `x != y` состояние `y` изменится, а следовательно, становится равным исходному значению `x`.

   В противном случае, если `x == y`, будет выполнена операция `y = 0 ^ {`*исходное* `y}`, и значение `y` не изменится.

   В любом случае `y` будет содержать исходное значение `x`.

3. `x = x ^ y`

   Или: `x = {0` *если исходные значения совпадают,* `1` *если различаются}* `^ {`*исходное* `x}`

   В этой точке `y` равно исходному значению `x`. Фактически эта строка эквивалентна предшествующей, но с другими переменными.

   Если выполнить `x = 1 ^ {`*исходное* `x}` для разных значений, состояние `x` изменится.

   Если выполнить `x = 0 ^ {`*исходное* `x}` для одинаковых значений, состояние `x` останется неизменным.

Эта операция выполняется с каждым битом. Так как она правильно меняет местами каждый бит, то в итоге она правильно поменяет местами два числа.

16.2. Напишите метод для определения относительной частоты вхождения слов в книге. А если алгоритм должен выполняться многократно?

РЕШЕНИЕ

Начнем с простого случая.

Решение. Однократный запрос

В этом случае мы просто просматриваем книгу пословно и считаем, сколько раз встречается конкретное слово. Это займет $O(n)$ времени. Лучших результатов добиться нельзя, так как необходимо проверить каждое слово в книге.

```
1  int getFrequency(String[] book, String word) {
2    word = word.trim().toLowerCase();
3    int count = 0;
4    for (String w : book) {
5      if (w.trim().toLowerCase().equals(word)) {
6        count++;
7     }
8    }
9    return count;
10 }
```

Также можно преобразовать строку к нижнему регистру и отсечь пропуски. Вы можете обсудить с интервьюером, насколько это необходимо (или желательно).

Решение. Повторяющиеся запросы

Если проверка будет выполняться многократно, можно потратить дополнительное время и память на предварительную обработку данных, например создать хеш-таблицу, которая будет устанавливать соответствие между словами и частотой их употребления. Тогда любое слово можно будет найти за $O(1)$ время. Код этого решения приведен ниже:

```
1  HashMap<String, Integer> setupDictionary(String[] book) {
2    HashMap<String, Integer> table =
3      new HashMap<String, Integer>();
4    for (String word : book) {
5      word = word.toLowerCase();
6      if (word.trim() != "") {
7        if (!table.containsKey(word)) {
8          table.put(word, 0);
9      }
10        table.put(word, table.get(word) + 1);
11     }
12   }
13   return table;
14 }
15
16 int getFrequency(HashMap<String, Integer> table, String word) {
17   if (table == null || word == null) return -1;
18   word = word.toLowerCase();
19   if (table.containsKey(word)) {
20     return table.get(word);
21   }
22   return 0;
23 }
```

Поскольку подобные задачи относительно просты, интервьюер будет особенно внимательно следить за тем, насколько аккуратно вы ее решите. Например, не забыли ли вы о проверке ошибок?

16.3. Для двух отрезков, заданных начальной и конечной точками, вычислите точку пересечения (если она существует).

РЕШЕНИЕ

Для начала необходимо решить, что же означает пересечение двух отрезков.

Чтобы две (бесконечные) линии пересекались, достаточно, чтобы они имели разные углы наклона. Если их углы наклона совпадают, то линии пересекаются при условии их совпадения. Чтобы пересекались два отрезка, приведенное выше условие должно выполняться, а также точка пересечения должна находиться в пределах каждого отрезка.

А если два отрезка лежат на одной линии? В этом случае необходимо убедиться в том, что они частично перекрываются. Если упорядочить отрезки по координатам x (начало перед концом, точка 1 перед точкой 2):

```
start1.x < start2.x && start1.x < end1.x && start2.x < end2.x
```

то пересечение присутствует только при выполнении условия:

start2 находится между start1 и end1

Теперь можно переходить к реализации этого алгоритма.

```
1  Point intersection(Point start1, Point end1, Point start2, Point end2) {
2    /* Упорядочить таким образом, чтобы в порядке значений x начальная
3     * точка предшествовала конечной, а точка 1 предшествовала 2. */
4    if (start1.x > end1.x) swap(start1, end1);
5    if (start2.x > end2.x) swap(start2, end2);
6    if (start1.x > start2.x) {
7      swap(start1, start2);
8      swap(end1, end2);
9    }
10
11   /* Вычисление линий (наклон и точка пересечения оси y). */
12   Line line1 = new Line(start1, end1);
13   Line line2 = new Line(start2, end2);
14
15   /* Если линии параллельны, они пересекаются только при совпадении
16    * yintercept, а start2 находится на линии 1. */
17   if (line1.slope == line2.slope) {
18     if (line1.yintercept == line2.yintercept &&
19         isBetween(start1, start2, end2)) {
20       return start2;
21     }
22     return null;
23   }
24
25   /* Вычисление координаты пересечения оси. */
26   double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
27   double y = x * line1.slope + line1.yintercept;
28   Point intersection = new Point(x, y);
29
30   /* Проверка принадлежности границам отрезка. */
31   if (isBetween(start1, intersection, end1) &&
32       isBetween(start2, intersection, end2)) {
33     return intersection;
34   }
35   return null;
36 }
37
38 /* Проверка того, что middle находится между start и end. */
39 boolean isBetween(double start, double middle, double end) {
40   if (start > end) {
41     return end <= middle && middle <= start;
42   } else {
43     return start <= middle && middle <= end;
44   }
45 }
46
47 /* Проверка того, что middle находится между start и end. */
48 boolean isBetween(Point start, Point middle, Point end) {
49   return isBetween(start.x, middle.x, end.x) &&
50          isBetween(start.y, middle.y, end.y);
51 }
52
53 /* Перестановка координат двух точек. */
```

```
54  void swap(Point one, Point two) {
55    double x = one.x;
56    double y = one.y;
57    one.setLocation(two.x, two.y);
58    two.setLocation(x, y);
59  }
60
61  public class Line {
62    public double slope, yintercept;
63
64    public Line(Point start, Point end) {
65    double deltaY = end.y - start.y;
66    double deltaX = end.x - start.x;
67    slope = deltaY / deltaX;
68    yintercept = end.y - slope * end.x;
69  }
70
71  public class Point {
72    public double x, y;
73    public Point(double x, double y) {
74      this.x = x;
75      this.y = y;
76    }
77
78    public void setLocation(double x, double y) {
79      this.x = x;
80      this.y = y;
81    }
82  }
```

Для простоты и компактности переменные класссов `Point` и `Line` объявлены открытыми (это заметно упрощает чтение кода). Вы можете обсудить достоинства и недостатки этого решения с интервьюером.

16.4. Разработайте алгоритм, проверяющий результат игры в крестики-нолики.

РЕШЕНИЕ

На первый взгляд эта задача кажется тривиальной — достаточно проверить доску. Разве это трудно? При ближайшем рассмотрении оказывается, что задача несколько сложнее и единственного «идеального» решения не существует. Выбор оптимального решения зависит от множества факторов.

Есть несколько принципиальных аспектов, которые следует учесть:

- Будет ли `hasWon` вызываться однократно или несколько раз (например, как компонент веб-сайта с онлайн-версией игры)? Во втором случае, возможно, стоит выделить время на предварительную обработку, чтобы оптимизировать время выполнения `hasWon`.
- Известен ли последний сделанный ход?
- Обычно в крестики-нолики играют на поле 3×3. Мы будем решать задачу для доски этого размера или попытаемся реализовать универсальное решение для поля размером $N \times N$?

- Что важнее: размер кода, скорость выполнения или понятность кода? Помните: самый эффективный код не всегда является лучшим. Очень важно, чтобы код был понятен и удобен в сопровождении.

Решение 1. Если hasWon вызывается много раз

Всего существует 3^9, или около 20 000, полей для игры в крестики-нолики (размером 3×3). Следовательно, игровое поле можно представить в формате `int`, где каждый разряд соответствует клетке (0 — пусто, 1 — крестик, 2 — нолик). Можно заранее создать хеш-таблицу или массив со всеми возможными расстановками в качестве ключей и значений, указывающих, кто победил. Тогда функция будет простой:

```
1  Piece hasWon(int board) {
2    return winnerHashtable[board];
3  }
```

Для преобразования игрового поля (представленного массивом символов) в `int` можно воспользоваться аналогом троичной системы счисления. Каждая доска представлена как $3^0 v_0 + 3^1 v_1 + 3^2 v_2 + \ldots + 3^8 v_8$, где $v_i = 0$, если клетка пустая, `1` — нолик, `2` — крестик.

```
1  enum Piece { Empty, Red, Blue };
2
3  int convertBoardToInt(Piece[][] board) {
4    int sum = 0;
5    for (int i = 0; i < board.length; i++) {
6      for (int j = 0; j < board[i].length; j++) {
7        /* С каждым значением в перечислении связывается целое число.
8         * Мы просто используем его. */
9        int value = board[i][j].ordinal();
10        sum = sum * 3 + value;
11     }
12    }
13    return sum;
14  }
```

Теперь нахождение победителя превращается в задачу поиска по хеш-таблице.

Конечно, если поле будет преобразовываться в этот формат каждый раз, когда потребуется найти победителя, это решение не даст совершенно никакой экономии времени. Но если поле будет храниться в таком формате с самого начала, процесс поиска станет чрезвычайно эффективным.

Решение 2. Если последний ход известен

Если последний сделанный ход известен (и победитель проверялся ранее), достаточно проверить только строку, столбец и диагональ, включающие позицию последнего хода.

```
1  Piece hasWon(Piece[][] board, int row, int column) {
2    if (board.length != board[0].length) return Piece.Empty;
3
4    Piece piece = board[row][column];
5
6    if (piece == Piece.Empty) return Piece.Empty;
```

```
7
8     if (hasWonRow(board, row) || hasWonColumn(board, column)) {
9       return piece;
10    }
11
12    if (row == column && hasWonDiagonal(board, 1)) {
13      return piece;
14    }
15
16    if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
17      return piece;
18    }
19
20    return Piece.Empty;
21  }
22
23  boolean hasWonRow(Piece[][] board, int row) {
24    for (int c = 1; c < board[row].length; c++) {
25      if (board[row][c] != board[row][0]) {
26        return false;
27      }
28    }
29    return true;
30  }
31
32  boolean hasWonColumn(Piece[][] board, int column) {
33    for (int r = 1; r < board.length; r++) {
34      if (board[r][column] != board[0][column]) {
35        return false;
36      }
37    }
38    return true;
39  }
40
41  boolean hasWonDiagonal(Piece[][] board, int direction) {
42    int row = 0;
43    int column = direction == 1 ? 0 : board.length - 1;
44    Piece first = board[0][column];
45    for (int i = 0; i < board.length; i++) {
46      if (board[row][column] != first) {
47        return false;
48      }
49      row += 1;
50      column += direction;
51    }
52    return true;
53  }
```

Из этой реализации можно исключить часть дублирующегося кода. Эта возможность будет продемонстрирована ниже.

Решение 3. Поле размером 3×3

Если задача решается только для поля 3×3, код получается относительно коротким и простым. Самое сложное — сделать его логичным и упорядоченным, исключив дублирование фрагментов.

Следующая реализация проверяет каждую строку, столбец и диагональ.

```
1  Piece hasWon(Piece[][] board) {
2    for (int i = 0; i < board.length; i++) {
3      /* Проверка строк */
4      if (hasWinner(board[i][0], board[i][1], board[i][2])) {
5        return board[i][0];
6      }
7
8      /* Проверка столбцов */
9      if (hasWinner(board[0][i], board[1][i], board[2][i])) {
10       return board[0][i];
11     }
12   }
13
14   /* Проверка диагонали */
15   if (hasWinner(board[0][0], board[1][1], board[2][2])) {
16     return board[0][0];
17   }
18
19   if (hasWinner(board[0][2], board[1][1], board[2][0])) {
20     return board[0][2];
21   }
22
23   return Piece.Empty;
24 }
25
26 boolean hasWinner(Piece p1, Piece p2, Piece p3) {
27   if (p1 == Piece.Empty) {
28     return false;
29   }
30   return p1 == p2 && p2 == p3;
31 }
```

Это нормальное решение, по которому относительно легко разобраться в происходящем. Проблема в том, что значения в нем жестко фиксированы, что повышает вероятность ошибки при вводе индексов.

Кроме того, это решение достаточно плохо масштабируется для игрового поля $N \times N$.

Решение 4. Поле размером $N \times N$

Существуют разные способы реализации алгоритма для игрового поля $N \times N$.

Вложенные циклы for

Самый очевидный подход — серия вложенных циклов `for`.

```
1  Piece hasWon(Piece[][] board) {
2    int size = board.length;
3    if (board[0].length != size) return Piece.Empty;
4    Piece first;
5
6    /* Проверка строк. */
7    for (int i = 0; i < size; i++) {
8      first = board[i][0];
9      if (first == Piece.Empty) continue;
10     for (int j = 1; j < size; j++) {
```

```
      if (board[i][j] != first) {
        break;
      } else if (j == size - 1) { // Последний элемент
        return first;
      }
     }
    }

    /* Проверка столбцов. */
    for (int i = 0; i < size; i++) {
      first = board[0][i];
      if (first == Piece.Empty) continue;
      for (int j = 1; j < size; j++) {
        if (board[j][i] != first) {
          break;
        } else if (j == size - 1) { // Последний элемент
          return first;
        }
      }
    }

    /* Проверка диагоналей. */
    first = board[0][0];
    if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
        if (board[i][i] != first) {
          break;
        } else if (i == size - 1) { // Последний элемент
          return first;
        }
      }
    }

    first = board[0][size - 1];
    if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
        if (board[i][size - i - 1] != first) {
          break;
        } else if (i == size - 1) { // Последний элемент
          return first;
        }
      }
    }

    return Piece.Empty;
  }
```

Решение, мягко говоря, не самое красивое: каждый раз проделывается практически одна и та же работа. Нужно поискать возможности повторного использования кода.

Увеличение и уменьшение

Чтобы повысить степень повторного использования кода, можно просто передать значения другой функции, которая увеличивает/уменьшает строки и столбцы. В этом случае функции `hasWon` потребуется только начальная позиция и величина приращения строки или столбца.

```
1  class Check {
2    public int row, column;
3    private int rowIncrement, columnIncrement;
4    public Check(int row, int column, int rowI, int colI) {
5      this.row = row;
6      this.column = column;
7      this.rowIncrement = rowI;
8      this.columnIncrement = colI;
9    }
10
11   public void increment() {
12     row += rowIncrement;
13     column += columnIncrement;
14   }
15
16   public boolean inBounds(int size) {
17     return row >= 0 && column >= 0 && row < size && column < size;
18   }
19 }
20
21 Piece hasWon(Piece[][] board) {
22   if (board.length != board[0].length) return Piece.Empty;
23   int size = board.length;
24
25   /* Построение списка для проверки. */
26   ArrayList<Check> instructions = new ArrayList<Check>();
27   for (int i = 0; i < board.length; i++) {
28     instructions.add(new Check(0, i, 1, 0));
29     instructions.add(new Check(i, 0, 0, 1));
30   }
31   instructions.add(new Check(0, 0, 1, 1));
32   instructions.add(new Check(0, size - 1, 1, -1));
33
34   /* Проверка. */
35   for (Check instr : instructions) {
36     Piece winner = hasWon(board, instr);
37     if (winner != Piece.Empty) {
38         return winner;
39    }
40   }
41   return Piece.Empty;
42 }
43
44 Piece hasWon(Piece[][] board, Check instr) {
45   Piece first = board[instr.row][instr.column];
46   while (instr.inBounds(board.length)) {
47     if (board[instr.row][instr.column] != first) {
48       return Piece.Empty;
49    }
50     instr.increment();
51   }
52   return first;
53 }
```

По сути класс Check выполняет функции итератора.

Итератор

Конечно, с таким же успехом можно воспользоваться обычным итератором.

```
1  Piece hasWon(Piece[][] board) {
2    if (board.length != board[0].length) return Piece.Empty;
3    int size = board.length;
4
5    ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
6    for (int i = 0; i < board.length; i++) {
7      instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
8      instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
9    }
10   instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
11   instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));
12
13   for (PositionIterator iterator : instructions) {
14     Piece winner = hasWon(board, iterator);
15     if (winner != Piece.Empty) {
16       return winner;
17     }
18   }
19   return Piece.Empty;
20 }
21
22 Piece hasWon(Piece[][] board, PositionIterator iterator) {
23   Position firstPosition = iterator.next();
24   Piece first = board[firstPosition.row][firstPosition.column];
25   while (iterator.hasNext()) {
26     Position position = iterator.next();
27     if (board[position.row][position.column] != first) {
28       return Piece.Empty;
29     }
30   }
31   return first;
32 }
33
34 class PositionIterator implements Iterator<Position> {
35   private int rowIncrement, colIncrement, size;
36   private Position current;
37
38   public PositionIterator(Position p, int rowIncrement,
39                           int colIncrement, int size) {
40     this.rowIncrement = rowIncrement;
41     this.colIncrement = colIncrement;
42     this.size = size;
43     current = new Position(p.row - rowIncrement, p.column - colIncrement);
44   }
45
46   @Override
47   public boolean hasNext() {
48     return current.row + rowIncrement < size &&
49            current.column + colIncrement < size;
50   }
51
52   @Override
```

```
53    public Position next() {
54      current = new Position(current.row + rowIncrement,
55                             current.column + colIncrement);
56      return current;
57    }
58  }
59
60  public class Position {
61    public int row, column;
62    public Position(int row, int column) {
63      this.row = row;
64      this.column = column;
65    }
66  }
```

Возможно, все это выходит за пределы необходимости, однако вам стоит обсудить возможные варианты с интервьюером. Цель этой задачи — оценить ваше умение писать стройный, логичный код, не создающий проблем с сопровождением.

16.5. Напишите алгоритм, вычисляющий число завершающих нулей в n!.

РЕШЕНИЕ

Самый простой подход — вычислить факториал и затем подсчитать, сколько конечных нулей присутствует в результате, повторяя операцию деления на 10. Однако существует одна трудность: факториал быстро выходит за границы допустимых значений `int`. Чтобы избежать этой проблемы, нужно рассмотреть задачу с точки зрения математики.

Возьмем факториал конкретного числа, например 19:

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

Конечный нуль появляется за счет умножения на 10, а умножение на 10 создается парой множителей 5 и 2.

Например, в 19! конечные нули создают следующие элементы:

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

Следовательно, чтобы подсчитать количество нулей, можно подсчитать количество пар произведений 5 и 2. Впрочем, множителей, кратных 2, будет существенно больше, чем кратных 5, поэтому достаточно ограничиться подсчетом умножений на 5.

Здесь есть один тонкий момент — число 15 кратно 5, а значит, добавляет один конечный 0, тогда как число 25 добавит два нуля, потому что 25 = 5×5.

Код можно написать двумя основными способами.

Во-первых, можно перебрать значения от 2 до n и подсчитать количество вхождений множителя 5 в каждое число.

```
1  /* Если в число входят множители 5, вернуть их количество.
2   * Например: 5 -> 1, 25-> 2, и т. д. */
3  int factorsOf5(int i) {
4    int count = 0;
5    while (i % 5 == 0) {
```

```
6       count++;
7       i /= 5;
8     }
9     return count;
10  }
11
12  int countFactZeros(int num) {
13    int count = 0;
14    for (int i = 2; i <= num; i++) {
15      count += factorsOf5(i);
16    }
17    return count;
18  }
```

Код работает, но задачу можно решить более эффективно. Сначала мы подсчитаем, сколько чисел, кратных 5, находится между 1 и `n` (`n/5`), а затем учтем количество умножений на 25 (`n/25`), на 125 и т. д.

Чтобы подсчитать, сколько раз `m` входит в `n`, нужно просто разделить `n` на `m`.

```
1  int countFactZeros(int num) {
2    int count = 0;
3    if (num < 0) {
4      return -1;
5    }
6    for (int i = 5; num / i > 0; i *= 5) {
7      count += num / i;
8    }
9    return count;
10 }
```

На первый взгляд задача напоминает головоломку, но к ее решению можно подойти логически. Проанализируйте задачу, выясните, за счет чего появляются нули, — и вы придете к решению. Чтобы не допустить ошибки в реализации, вам следует четко сформулировать правила.

16.6. Для двух целочисленных массивов найдите пару значений (по одному значению из каждого массива) с минимальной (неотрицательной) разностью. Верните эту разность.

Пример:

Ввод: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Вывод: 3 для пары (11, 8).

РЕШЕНИЕ

Метод «грубой силы»

Простейшее решение просто перебирает все пары, вычисляет разность текущей пары и сравнивает ее с текущей минимальной разностью.

```
1  int findSmallestDifference(int[] array1, int[] array2) {
2    if (array1.length == 0 || array2.length == 0) return -1;
```

```
4    int min = Integer.MAX_VALUE;
5    for (int i = 0; i < array1.length; i++) {
6      for (int j = 0; j < array2.length; j++) {
7        if (Math.abs(array1[i] - array2[j]) < min) {
8          min = Math.abs(array1[i] - array2[j]);
9        }
10       }
11     }
12     return min;
13   }
```

Одна из незначительных оптимизаций, которую можно было бы применить, — немедленно вернуть управление при обнаружении разности 0, потому что это наименьшая возможная разность. Впрочем, в зависимости от входных данных такое решение может работать даже медленнее.

Оно работает быстрее только в том случае, если пара с нулевой разностью занимает одну из начальных позиций в списке пар. Но для добавления этой оптимизации каждый раз приходится выполнять лишнюю строку кода. Таким образом, оптимизация оказывается неоднозначной; при одних данных она ускоряет работу, а при других замедляет. А так как чтение кода усложняется, возможно, от этой оптимизации лучше отказаться.

С этой оптимизацией или без нее алгоритм выполняется за время $O(AB)$.

Оптимизированное решение

Более эффективное решение основано на предварительной сортировке массивов. После того как массивы будут отсортированы, минимальная разность находится простым перебором.

Возьмем следующие два массива:

A: {1, 2, 11, 15}

B: {4, 12, 19, 23, 127, 235}

Опробуем следующий подход:

1. Допустим, указатель `a` указывает на начало `A`, а указатель `b` указывает на начало `B`. Текущая разность `a` и `b` равна 3. Сохраняем ее в `min`.
2. Как (теоретически) уменьшить эту разность? Значение `b` больше, чем значение `a`, так что перемещение `b` только увеличит разность. Следовательно, перемещать нужно `a`.
3. Теперь `a` указывает на 2, а `b` (все еще) указывает на 4. Разность равна 2, значение `min` необходимо обновить. Перемещаем `a`, так как это значение меньше.
4. Сейчас `a` указывает на 11, а `b` указывает на 4. Перемещаем `b`.
5. Теперь `a` указывает на 11, а `b` указывает на 12. Обновляем `min` значением 1, перемещаем `b`.

И так далее.

```
1  int findSmallestDifference(int[] array1, int[] array2) {
2    Arrays.sort(array1);
3    Arrays.sort(array2);
4    int a = 0;
```

```
5    int b = 0;
6    int difference = Integer.MAX_VALUE;
7    while (a < array1.length && b < array2.length) {
8      if (Math.abs(array1[a] - array2[b]) < difference) {
9        difference = Math.abs(array1[a] - array2[b]);
10     }
11
12      /* Переместить меньшее значение. */
13      if (array1[a] < array2[b]) {
14        a++;
15      } else {
16        b++;
17     }
18    }
19    return difference;
20  }
```

Алгоритм требует времени $O(A \log A + B \log B)$ для выполнения сортировки, и времени $O(A + B)$ для нахождения минимальной разности. Следовательно, общее время выполнения равно $O(A \log A + B \log B)$.

16.7. Напишите метод, находящий максимальное из двух чисел без использования `if-else` или любых других операторов сравнения.

РЕШЕНИЕ

Самый распространенный вариант реализации функции `max` — проверка знака выражения `a - b`. В этом случае мы не можем использовать операторы сравнения, но *можем* использовать умножение.

Обозначим знак выражения `a - b` как `k`. Если `a - b >= 0`, то `k = 1`, иначе `k = 0`. Пусть `q` будет инвертированным значением `k`.

Код будет иметь вид:

```
1  /* 1 переводится в 0, а 0 переводится в 1 */
2  int flip(int bit) {
3    return 1^bit;
4  }
5
6  /* Возвращает 1 для положительных a, или 0 для отрицательных */
7  int sign(int a) {
8    return flip((a >> 31) & 0x1);
9  }
10
11  int getMaxNaive(int a, int b) {
12    int k = sign(a - b);
13    int q = flip(k);
14    return a * k + b * q;
15  }
```

Код почти работает. К сожалению, при переполнении `a - b` начинаются проблемы. Предположим, что `a = INT_MAX - 2` и `b = -15`. В этом случае `a - b` окажется больше `INT_MAX` и вызовет переполнение (значение станет отрицательным).

Можно использовать тот же подход, но придумать другую реализацию. Нам нужно, чтобы выполнялось условие k = 1, когда a > b. Для этого придется использовать более сложную логику.

Когда возникает переполнение a - b? Только тогда, когда a — положительное число, а b — отрицательное (или наоборот). Трудно обнаружить факт переполнения, но мы в состоянии определить, что a и b имеют разные знаки. Если у a и b разные знаки, то k должно быть равно sign(a).

Логика будет следующей:

```
1  Если a и b имеют разные знаки:
2    // если a > 0, то b < 0, и k = 1.
3    // если a < 0, то b > 0, и k = 0.
4    // В любом случае k = sign(a)
5    let k = sign(a)
6  else
7    let k = sign(a - b) // Переполнение невозможно
```

Следующая реализация использует умножение вместо конструкций if:

```
1  int getMax(int a, int b) {
2    int c = a - b;
3
4    int sa = sign(a); // если a >= 0, то 1, иначе 0
5    int sb = sign(b); // если b >= 0, то 1, иначе 0
6    int sc = sign(c); // зависит от того, приводит ли к переполнению a-b
7
8    /* Цель: определить значение k, равное 1, если a > b, или 0 при a < b.
9     * (если a = b, то значение k несущественно) */
10
11   // Если у a и b разные знаки, то k = sign(a)
12   int use_sign_of_a = sa ^ sb;
13
14   // Если у a и b одинаковый знак, то k = sign(a - b)
15   int use_sign_of_c = flip(sa ^ sb);
16
17   int k = use_sign_of_a * sa + use_sign_of_c * sc;
18   int q = flip(k); // Противоположность k
19
20   return a * k + b * q;
21 }
```

Отметим, что для большей наглядности мы разделяем код на множество методов и переменных. Конечно, это не самый компактный или эффективный способ записи, но код становится намного понятнее.

16.8. Задано целое число. Выведите его значение в текстовом виде (например, «одна тысяча двести тридцать четыре»).

РЕШЕНИЕ

Это несложная, но довольно утомительная задача. Важно применить организованный подход к решению и подготовить хороший набор тестовых сценариев.

Можно преобразовать число вроде 19 323 984 в несколько трехразрядных сегментов с соответствующими добавками «thousand» и «million». Например[1]:

```
convert(19,323,984) = convert(19) + " million " +
                      convert(323) + " thousand " +
                      convert(984)
```

Следующий код реализует этот алгоритм.

```
1  String[] smalls = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
2    "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
3    "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
4  String[] tens = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty",
"Seventy",
5    "Eighty", "Ninety"};
6  String[] bigs = {"", "Thousand", "Million", "Billion"};
7  String hundred = "Hundred";
8  String negative = "Negative";
9
10 String convert(int num) {
11   if (num == 0) {
12     return smalls[0];
13   } else if (num < 0) {
14     return negative + " " + convert(-1 * num);
15   }
16
17   LinkedList<String> parts = new LinkedList<String>();
18   int chunkCount = 0;
19
20   while (num > 0) {
21     if (num % 1000 != 0) {
22       String chunk = convertChunk(num % 1000) + " " + bigs[chunkCount];
23       parts.addFirst(chunk);
24     }
25     num /= 1000; // Сдвиг группы
26     chunkCount++;
27   }
28
29   return listToString(parts);
30 }
31
32 String convertChunk(int number) {
33   LinkedList<String> parts = new LinkedList<String>();
34
35   /* Преобразование сотен */
36   if (number >= 100) {
37     parts.addLast(smalls[number / 100]);
38     parts.addLast(hundred);
39     number %= 100;
40   }
41
42   /* Преобразование десятков */
```

[1] Листинг приводится без перевода — как в оригинале, чтобы не переписывать весь код. Для создания аналогичной русскоязычной программы код придется несколько изменить. — *Примеч. пер.*

```
43    if (number >= 10 && number <= 19) {
44      parts.addLast(smalls[number]);
45    } else if (number >= 20) {
46      parts.addLast(tens[number / 10]);
47      number %= 10;
48    }
49
50    /* Преобразование единиц */
51    if (number >= 1 && number <= 9) {
52      parts.addLast(smalls[number]);
53    }
54
55    return listToString(parts);
56  }
57  /* Преобразование связного списка в строку с разделением пробелами. */
58  String listToString(LinkedList<String> parts) {
59    StringBuilder sb = new StringBuilder();
60    while (parts.size() > 1) {
61      sb.append(parts.pop());
62      sb.append(" ");
63    }
64    sb.append(parts.pop());
65    return sb.toString();
66  }
```

Главное в решении этой задачи — тщательная проверка всех особых случаев. В этой задаче их много.

16.9. Напишите методы, реализующие операции умножения, вычитания и деления целых чисел. Результатом во всех случаях должно быть целое число. В коде разрешается использовать только оператор сложения.

РЕШЕНИЕ

Итак, разрешена только одна операция: сложение. В подобных задачах полезно вспомнить суть математических операций и как их можно реализовать с помощью сложения (или уже смоделированных операций).

Вычитание

Как реализовать вычитание с помощью сложения? Это предельно просто. Операция `a - b` — то же самое, что и `a + (-1) * b`. Впрочем, оператор умножения недоступен, поэтому придется создать функцию `negate`.

```
1  /* Изменение знака числа. */
2  int negate(int a) {
3    int neg = 0;
4    int newSign = a < 0 ? 1 : -1;
5    while (a != 0) {
6      neg += newSign;
7      a += newSign;
8    }
9    return neg;
10 }
11
```

```
12  /* Вычитание выполняется изменением знака b и суммированием */
13  int minus(int a, int b) {
14    return a + negate(b);
15  }
```

Изменение знака k реализуется прибавлением –1 k раз. Заметим, что эта реализация выполняется за время $O(k)$.

Если эффективности уделяется особое внимание, можно попытаться довести a до нуля быстрее (в следующем объяснении предполагается, что a положительно). Для этого a сначала уменьшается на 1, затем на 2, затем на 4, на 8 и т. д. Назовем это значение `delta`. Нужно, чтобы значение a уменьшилось ровно до нуля. Если уменьшение a на следующий шаг `delta` приведет к изменению знака a, значение `delta` сбрасывается в 1, и процесс повторяется.

Пример:

```
a:       29   28   26   22   14   13   11    7    6    4    0
delta:   -1   -2   -4   -8   -1   -2   -4   -1   -2   -4
```

Ниже приведена реализация этого алгоритма.

```
1  int negate(int a) {
2    int neg = 0;
3    int newSign = a < 0 ? 1 : -1;
4    int delta = newSign;
5    while (a != 0) {
6      boolean differentSigns = (a + delta > 0) != (a > 0);
7      if (a + delta != 0 && differentSigns) { // Сброс delta
8        delta = newSign;
9      }
10       neg += delta;
11       a += delta;
12       delta += delta; // Удвоение delta
13     }
14     return neg;
15   }
```

Существуют и более эффективные решения. Например, вместо того чтобы сбрасывать `delta` в 1, можно вернуть предыдущее значение. По сути `delta` сначала будет «подниматься» на степень 2, а потом «опускаться» на степень 2. Такое решение будет выполняться за время $O(\log a)$. С другой стороны, оно потребует применения стека, деления или поразрядного сдвига — все это может оказаться нарушением духа исходной задачи. Впрочем, вы можете обсудить эти реализации с интервьюером.

Умножение

Связь между сложением и умножением тоже достаточно очевидна. Чтобы перемножить a и b, нужно сложить значение a с самим собой b раз.

```
1  /* Умножение a на b реализуется сложением a с собой b раз */
2  int multiply(int a, int b) {
3    if (a < b) {
4      return multiply(b, a); // алгоритм работает быстрее, если b < a
5    }
6    int sum = 0;
7    for (int i = abs(b); i > 0; i = minus(i, 1)) {
```

```
8       sum += a;
9     }
10    if (b < 0) {
11      sum = negate(sum);
12    }
13    return sum;
14  }
15
16  /* Возвращает модуль числа */
17  int abs(int a) {
18    if (a < 0) {
19      return negate(a);
20    } else {
21      return a;
22    }
23  }
```

Единственное, на что следует обратить внимание в этом коде, — умножение отрицательных чисел. Если `b` — отрицательное число, то необходимо изменить знак суммы: `multiply(a, b) <-- abs(b) * a * (-1 if b < 0)`.

Для решения этой задачи была создана вспомогательная функция `abs`.

Деление

Бесспорно, самая сложная из математических операций — деление. К счастью, в реализации метода `divide` разрешено использовать методы `multiply`, `subtract` и `negate`.

Нам нужно найти `x = a / b`. Или в эквивалентной формулировке — найти такое значение `x`, для которого `a = bx`. Теперь мы изменили условие так, чтобы задачу можно было решить с помощью уже известной нам операции — умножения.

Можно решить задачу, умножая `b` на все увеличивающиеся числа, до тех пор пока не достигнем `a`. Это очень неэффективный способ, поскольку каждая операция умножения состоит из множества операций сложения.

Еще раз взглянув на уравнение `a = xb`, мы видим, что `x` можно вычислить многократным суммированием `b`, пока не будет получено `a`. Количество экземпляров `b`, необходимых для получения `a`, и будет искомой величиной `x`.

Конечно, это решение нельзя назвать делением, но оно работает. Целочисленное деление, которое нам было предложено реализовать, и так должно округлять результат.

Приведенный ниже код реализует данный алгоритм:

```
1  int divide(int a, int b) throws java.lang.ArithmeticException {
2    if (b == 0) {
3      throw new java.lang.ArithmeticException("ERROR");
4    }
5    int absa = abs(a);
6    int absb = abs(b);
7
8    int product = 0;
9    int x = 0;
10   while (product + absb <= absa) { /* Не проходить дальше a */
11     product += absb;
12     x++;
13   }
```

```
14
15    if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
16      return x;
17    } else {
18      return negate(x);
19    }
20  }
```

При решении этой задачи необходимо учитывать следующие аспекты:

- Подойдите к решению логическим путем: вспомните, что делает умножение и деление. Не забывайте: все (хорошие) задачи на собеседованиях можно решить логическим, методичным способом.
- Это отличная возможность продемонстрировать свое умение писать чистый и понятный код, пригодный для повторного использования. Если вы решаете эту задачу самостоятельно, но не выделили `negate` в отдельный метод, сделайте это, как только увидите, что код используется в нескольких местах.
- Будьте осторожны в своих предположениях. Например, не следует полагать, что все числа положительны или что `a > b`.

16.10. Имеется список людей с годами рождения и смерти. Напишите метод для вычисления года, в котором количество живых людей из списка было максимальным. Предполагается, что все люди родились в промежутке от 1900 до 2000 г. (включительно). Если человек прожил хотя бы часть года, он включается в счетчик этого года. Например, человек, родившийся в 1908 г. и умерший в 1909 г., учитывается как в 1908-м, так и в 1909 г.

РЕШЕНИЕ

Для начала следует составить общее представление о том, как должно выглядеть решение. В формулировке задачи не указана точная форма входных данных. В реальном собеседовании можно уточнить структуру входных данных у интервьюера. Также вы можете явно изложить свои (разумные) предположения.

Здесь мы сделаем свои предположения. Будем считать, что входные данные передаются в массиве простых объектов `Person`:

```
1  public class Person {
2    public int birth;
3    public int death;
4    public Person(int birthYear, int deathYear) {
5      birth = birthYear;
6      death = deathYear;
7    }
8  }
```

Также можно включить в класс `Person` методы `getBirthYear()` и `getDeathYear()`. Возможно, это улучшит стиль программирования, но для компактности и ясности мы просто объявляем переменные открытыми.

Здесь важен сам факт использования объекта `Person`. Тем самым вы продемонстрируете лучший стиль, чем, скажем, при использовании двух целочисленных

массивов, для годов рождения и смерти (с неявным предположением о том, что элементы `births[i]` и `deaths[i]` соответствуют одному человеку). Возможности продемонстрировать хороший стиль программирования встречаются не так часто, и их не стоит упускать.

Учитывая сказанное, начнем с алгоритма, действующего методом «грубой силы».

Метод «грубой силы»

Алгоритм «грубой силы» напрямую следует из формулировки задачи. Нужно найти год с максимальным количеством живущих людей. Соответственно, мы перебираем данные за каждый год и проверяем, сколько живущих людей в этом году.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int maxAlive = 0;
3    int maxAliveYear = min;
4
5    for (int year = min; year <= max; year++) {
6      int alive = 0;
7      for (Person person : people) {
8        if (person.birth <= year && year <= person.death) {
9          alive++;
10       }
11     }
12      if (alive > maxAlive) {
13        maxAlive = alive;
14        maxAliveYear = year;
15     }
16    }
17
18    return maxAliveYear;
19  }
```

Обратите внимание на передачу значений наименьшего года (1900) и наибольшего года (2000). Эти значения не должны жестко фиксироваться в коде.

Время выполнения составляет $O(RP)$, где R — диапазон лет (100 в данном случае), а P — количество людей.

Слегка улучшенное решение методом «грубой силы»

Другое, слегка улучшенное решение основано на создании массива для отслеживания количества людей, родившихся в каждый год. Затем мы перебираем список людей и увеличиваем элемент массива для каждого года, в котором этот человек жив.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int[] years = createYearMap(people, min, max);
3    int best = getMaxIndex(years);
4    return best + min;
5  }
6
7  /* Сохранение информации о годах жизни в массиве. */
8  int[] createYearMap(Person[] people, int min, int max) {
9    int[] years = new int[max - min + 1];
10    for (Person person : people) {
11      incrementRange(years, person.birth - min, person.death - min);
12    }
```

```
13    return years;
14  }
15
16  /* Увеличение массива для каждого значения от left до right. */
17  void incrementRange(int[] values, int left, int right) {
18    for (int i = left; i <= right; i++) {
19      values[i]++;
20    }
21  }
22
23  /* Получение индекса наибольшего элемента в массиве. */
24  int getMaxIndex(int[] values) {
25    int max = 0;
26    for (int i = 1; i < values.length; i++) {
27      if (values[i] > values[max]) {
28        max = i;
29     }
30    }
31    return max;
32  }
```

Будьте внимательны с размером массива в строке 9. Если диапазон состоит из годов с 1900-го до 2000-го включительно, то он содержит 101 элемент, а не 100. Вот почему размер массива вычисляется по формуле `max - min + 1`.

Чтобы оценить время выполнения, разобьем код на части.

- Сначала создается массив размером R, где R — разность между `max` и `min`.
- Затем для P людей перебираются годы (Y), в которые этот человек живет.
- Затем массив размером R перебирается снова.

Общее время выполнения составляет $O(PY + R)$. В худшем случае $Y = R$, и никаких улучшений по сравнению с первым алгоритмом не будет.

Более эффективное решение

Давайте создадим пример. (Собственно, пример пригодится практически в любой задаче — в идеале вы уже сами это сделали.) Каждый столбец в следующей таблице соответствует одному человеку. Для компактности мы ограничились двумя последними цифрами каждого года.

```
рождение: 12  20  10  01  10  23  13  90  83  75
смерть:   15  90  98  72  98  82  98  98  99  94
```

Стоит заметить, что соответствие между этими годами на самом деле не так уж важно. Каждое рождение добавляет, а каждая смерть исключает одного человека.

Так как соответствия не важны, отсортируем оба массива — это поможет в решении задачи.

```
рождение: 01  10  10  12  13  20  23  75  83  90
смерть:   15  72  82  90  94  98  98  98  98  99
```

Переберем эти годы.

- В год 0 никто не живет.
- В год 1 рождается один человек.

- В годы 2–9 не происходит ничего.
- В год 10 рождаются два человека. На этот момент живы трое.
- В год 15 один человек умирает, в живых остаются двое.

И так далее.

При таком способе перебора массива мы сможем отслеживать количество живых людей в каждом году.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int[] births = getSortedYears(people, true);
3    int[] deaths = getSortedYears(people, false);
4
5    int birthIndex = 0;
6    int deathIndex = 0;
7    int currentlyAlive = 0;
8    int maxAlive = 0;
9    int maxAliveYear = min;
10
11    /* Перебор элементов массивов. */
12    while (birthIndex < births.length) {
13      if (births[birthIndex] <= deaths[deathIndex]) {
14        currentlyAlive++; // добавление рождения
15        if (currentlyAlive > maxAlive) {
16          maxAlive = currentlyAlive;
17          maxAliveYear = births[birthIndex];
18      }
19        birthIndex++; // смещение индекса рождений
20      } else if (births[birthIndex] > deaths[deathIndex]) {
21        currentlyAlive--; // исключение смерти
22        deathIndex++; // смещение индекса смертей
23     }
24    }
25
26    return maxAliveYear;
27  }
28
29  /* Копирование дат рождения или смерти (в зависимости от copyBirthYear)
30   * в целочисленный массив с последующей сортировкой. */
31  int[] getSortedYears(Person[] people, boolean copyBirthYear) {
32    int[] years = new int[people.length];
33    for (int i = 0; i < people.length; i++) {
34      years[i] = copyBirthYear ? people[i].birth : people[i].death;
35    }
36    Arrays.sort(years);
37    return years;
38  }
```

В строке 13 необходимо тщательно подумать над тем, какой оператор должен использоваться: < или <=. Проблемы создаст сценарий, в котором даты рождения и смерти приходятся на один год (неважно, относятся ли они к одному человеку или к разным). В такой ситуации рождение должно быть включено перед смертью, чтобы человек считался живым в этот год; по этой причине в строке 13 используется оператор <=.

Также необходимо тщательно выбрать место обновления `maxAlive` и `maxAliveYear`. Оно должно следовать после `currentAlive++`, чтобы в нем учитывалась обновленная сумма. С другой стороны, оно должно предшествовать `birthIndex++`, или год окажется неправильным.

Алгоритм выполняется за время $O(P \log P)$, где P — количество людей.

Дальнейшая оптимизация

Можно ли улучшить и это решение? Избавимся от сортировки и вернемся к работе с несортированными значениями:

```
рождение: 12  20  10  01  10  23  13  90  83  75
смерть:   15  90  98  72  98  82  98  98  99  94
```

Ранее упоминалось о том, что рождение — это просто добавление, а смерть — вычитание одного человека. Представим данные в этой логике:

```
01: +1    10: +1    10: +1    12: +1    13: +1
15: -1    20: +1    23: +1    72: -1    75: +1
82: -1    83: +1    90: +1    90: -1    94: -1
98: -1    98: -1    98: -1    98: -1    99: -1
```

Можно создать массив, в котором элемент *массив*[*год*] обозначает изменение населения за этот год. Чтобы создать такой массив, мы переберем список людей и будем увеличивать значение, когда люди рождаются, и уменьшать его, когда люди умирают.

Получив массив, мы переберем все годы с отслеживанием текущего населения (каждый раз прибавляя значение *массив*[*год*]).

Логика выглядит разумно, но в ней стоит разобраться повнимательнее. Работает ли она?

Первый граничный случай, который следует рассмотреть, — когда человек умирает в год своего рождения. Операции увеличения и уменьшения компенсируются, и прирост населения считается нулевым. Однако согласно формулировке задачи человек в этот год должен учитываться как живущий.

В действительности проблема еще шире: она распространяется на всех людей. Люди, умершие в 1908 г., не должны исключаться из населения до 1909 г.

Проблема решается просто: вместо уменьшения `array[deathYear]` следует уменьшать `array[deathYear + 1]`.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    /* Построение массива прироста населения. */
3    int[] populationDeltas = getPopulationDeltas(people, min, max);
4    int maxAliveYear = getMaxAliveYear(populationDeltas);
5    return maxAliveYear + min;
6  }
7
8  /* Добавление годов рождения и смерти в массив прироста. */
9  int[] getPopulationDeltas(Person[] people, int min, int max) {
10   int[] populationDeltas = new int[max - min + 2];
11   for (Person person : people) {
12     int birth = person.birth - min;
13     populationDeltas[birth]++;
14
```

```
15      int death = person.death - min;
16      populationDeltas[death + 1]--;
17    }
18    return populationDeltas;
19  }
20
21  /* Вычисление текущих сумм и возвращение индекса с максимумом. */
22  int getMaxAliveYear(int[] deltas) {
23    int maxAliveYear = 0;
24    int maxAlive = 0;
25    int currentlyAlive = 0;
26    for (int year = 0; year < deltas.length; year++) {
27      currentlyAlive += deltas[year];
28      if (currentlyAlive > maxAlive) {
29        maxAliveYear = year;
30        maxAlive = currentlyAlive;
31     }
32    }
33
34    return maxAliveYear;
35  }
```

Алгоритм выполняется за время $O(R + P)$, где R — диапазон лет, а P — количество людей. Хотя $O(R + P)$ может быть быстрее $O(P \log P)$ для многих ожидаемых вариантов ввода, напрямую сравнивать эти скорости нельзя.

16.11. Вы строите трамплин для прыжков в воду, складывая деревянные планки концом к концу. Планки делятся на два типа: длиной `shorter` и длиной `longer`. Необходимо использовать ровно K планок. Напишите метод, генерирующий все возможные длины трамплина.

РЕШЕНИЕ

Как взяться за решение этой задачи? Задумайтесь над тем, какие решения принимаются при построении трамплина. Этот анализ приведет нас к рекурсивному алгоритму.

Рекурсивное решение

Процесс построения трамплина состоит из K решений; каждый раз выбирается следующая доска. После размещения K досок получается законченный трамплин, который можно добавить в список (при условии, что эта длина еще не встречалась ранее).

Руководствуясь этой логикой, можно написать рекурсивный код. Обратите внимание: отслеживать последовательность досок не нужно, достаточно знать текущую длину и количество оставшихся досок.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    getAllLengths(k, 0, shorter, longer, lengths);
4    return lengths;
5  }
6
7  void getAllLengths(int k, int total, int shorter, int longer,
```

```
8                     HashSet<Integer> lengths) {
9    if (k == 0) {
10     lengths.add(total);
11     return;
12   }
13   getAllLengths(k - 1, total + shorter, shorter, longer, lengths);
14   getAllLengths(k - 1, total + longer, shorter, longer, lengths);
15 }
```

Каждая длина добавляется в коллекцию HashSet, автоматически предотвращающую добавление дубликатов. Алгоритм выполняется за время $O(2^K)$, потому что при каждом рекурсивном вызове существуют два варианта, а рекурсия осуществляется на глубину K.

Решение на базе мемоизации

Этот алгоритм, как и многие рекурсивные алгоритмы (особенно имеющие экспоненциальное время выполнения), можно оптимизировать применением мемоизации (разновидности динамического программирования).

Отметим, что некоторые рекурсивные вызовы оказываются практически эквивалентными. Например, выбор доски 1 с последующим выбором доски 2 эквивалентен выбору доски 2 с последующим выбором доски 1.

Следовательно, если пара (*сумма, количество досок*) уже встречалась ранее, путь рекурсии можно прервать. Для этого можно воспользоваться HashSet с ключом (*сумма, количество досок*).

Многие кандидаты допускают в этом месте ошибку. Вместо того чтобы останавливаться только на встречавшейся ранее комбинации *(сумма, количество досок)*, они останавливаются при обнаружении суммы. Это неправильно — две доски длиной 1 не эквивалентны одной доске длиной 2 из-за разного количества оставшихся досок. В задачах мемоизации всегда следует быть очень осторожными с выбором ключа.

Код этого решения очень похож на предыдущую версию.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    HashSet<String> visited = new HashSet<String>();
4    getAllLengths(k, 0, shorter, longer, lengths, visited);
5    return lengths;
6  }
7
8  void getAllLengths(int k, int total, int shorter, int longer,
9                     HashSet<Integer> lengths, HashSet<String> visited) {
10   if (k == 0) {
11     lengths.add(total);
12     return;
13   }
14   String key = k + " " + total;
15   if (visited.contains(key)) {
16     return;
17   }
18   getAllLengths(k - 1, total + shorter, shorter, longer, lengths, visited);
19   getAllLengths(k - 1, total + longer, shorter, longer, lengths, visited);
20   visited.add(key);
21 }
```

Для простоты мы используем в качестве ключа строковое представление суммы и текущего количества досок. Возможно, кто-то посчитает, что для представления пары лучше использовать структуру данных. У такого решения есть свои преимущества, но есть и недостатки. Упомяните о них в общении с интервьюером.

Рассчитать время выполнения этого алгоритма не так просто. Для начала можно понять, что мы по сути заполняем таблицу СУММЫ × КОЛИЧЕСТВО ДОСОК. Наибольшая возможная сумма равна K * LONGER, а наибольшее возможное количество досок равно K. Следовательно, время выполнения будет не хуже $O(K^2 *$ LONGER$)$.

Сколько уникальных сумм можно получить?

Заметим, что любые пути с одинаковым количеством досок каждого типа будут иметь одинаковые суммы. Так как количество досок каждого типа не может превышать K, возможны только K разных сумм. Следовательно, таблица в действительности имеет размеры $K \times K$, а время выполнения составит $O(K^2)$.

Оптимальное решение

Если перечитать предыдущий абзац, можно заметить нечто интересное. Существуют всего K разных сумм. Разве не в этом заключается суть задачи — найти все возможные суммы?

На самом деле находить все возможные последовательности досок не нужно. Достаточно перебрать все уникальные множества из K досок (множества, не порядки!). При двух типах досок существуют всего K способов выбора K досок: {0 типа A, K типа B}, {1 типа A, $K-1$ типа B}, {2 типа A, $K-2$ типа B}, ...

Перебор можно выполнить в простом цикле `for`. Для каждой «последовательности» просто вычисляется сумма:

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    for (int nShorter = 0; nShorter <= k; nShorter++) {
4      int nLonger = k - nShorter;
5      int length = nShorter * shorter + nLonger * longer;
6      lengths.add(length);
7    }
8    return lengths;
9  }
```

Класс `HashSet` используется для сохранения единства с предыдущими решениями. На самом деле он здесь не обязателен, потому что дубликатов быть не должно. Вместо него можно использовать `ArrayList`. Однако в этом случае необходимо обработать особую ситуацию с двумя типами досок одинаковой длины (тогда просто возвращается `ArrayList` размером 1).

16.12. Разметка XML занимает слишком много места. Закодируйте ее так, чтобы каждый тег заменялся заранее определенным целым значением:

```
Элемент --> Тег Атрибуты END ДочерниеТеги END
Атрибут --> Тег Значение
      END --> 0
      Тег --> любое предопределенное целое значение
    Значение --> строка значение END
```

Для примера преобразуем в сжатую форму следующую разметку XML. Предполагается, что `family -> 1`, `person -> 2`, `firstName -> 3`, `lastName -> 4`, `state -> 5`:

```
<family lastName="McDowell" state="CA">
    <person firstName="Gayle">Some Message</person>
</family>
```

После преобразования код будет иметь вид:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Напишите код, выводящий закодированную версию XML-элемента (переданного в объектах `Element` и `Attributes`).

РЕШЕНИЕ

Поскольку мы знаем, что элемент передается в объектах **`Element`** и **`Attribute`**, код получается достаточно простым. При реализации решения мы используем древовидную структуру.

В процессе работы для элементов структуры XML будет неоднократно вызываться `encode()`, обрабатывающий код по-разному в зависимости от типа элемента XML.

```
1  void encode(Element root, StringBuilder sb) {
2    encode(root.getNameCode(), sb);
3    for (Attribute a : root.attributes) {
4      encode(a, sb);
5    }
6    encode("0", sb);
7    if (root.value != null && root.value != "") {
8      encode(root.value, sb);
9    } else {
10     for (Element e : root.children) {
11       encode(e, sb);
12     }
13   }
14   encode("0", sb);
15 }
16
17 void encode(String v, StringBuilder sb) {
18   sb.append(v);
19   sb.append(" ");
20 }
21
22 void encode(Attribute attr, StringBuilder sb) {
23   encode(attr.getTagCode(), sb);
24   encode(attr.value, sb);
25 }
26
27 String encodeToString(Element root) {
28   StringBuilder sb = new StringBuilder();
29   encode(root, sb);
30   return sb.toString();
31 }
```

Обратите внимание на очень простой метод `encode` (в строке 17). На самом деле этот метод не обязателен; он всего лишь вставляет строку и пробел после нее. С другой стороны, он гарантирует, что после каждого элемента будет вставлен пробел. Без него легко нарушить процесс декодирования, просто забыв вставить пустую строку.

16.13. Для двух квадратов на плоскости найдите линию, которая делит эти два квадрата пополам. Предполагается, что верхняя и нижняя стороны квадрата проходят параллельно оси x.

РЕШЕНИЕ

Прежде чем начать, нам нужно подумать, что понимается под «линией». Как будет задаваться линия — углом наклона и точкой пересечения с осью y? Двумя точками на линии? Или под линией на самом деле понимается отрезок, начало и конец которого лежат на сторонах квадратов?

Чтобы задача была более интересной, мы выберем третий вариант — отрезок, начало и конец которого лежат на сторонах квадратов. Находясь на собеседовании, вы можете обсудить данный момент с интервьюером.

Прямая, которая делит два квадрата пополам, должна проходить через их середины. Наклон прямой описывается формулой: `slope = (y1 - y2) / (x1 - x2)`. Этим же принципом можно руководствоваться, чтобы рассчитать начальную и конечную точки отрезка.

В следующем коде предполагается, что начало координат (0, 0) находится в левом верхнем углу.

```
1  public class Square {
2    ...
3    public Point middle() {
4      return new Point((this.left + this.right) / 2.0,
5                       (this.top + this.bottom) / 2.0);
6    }
7
8    /* Возвращает точку, в которой отрезок, соединяющий mid1 и mid2,
9     * пересекает сторону квадрата 1. То есть мы проводим линию из mid2
10    * в mid1 и продолжаем ее до стороны квадрата. */
11    public Point extend(Point mid1, Point mid2, double size) {
12      /* Определение направления, в котором идет линия mid2 -> mid1. */
13      double xdir = mid1.x < mid2.x ? -1 : 1;
14      double ydir = mid1.y < mid2.y ? -1 : 1;
15
16      /* Если у mid1 и mid2 значения x совпадают, при вычислении наклона
17       * произойдет деление на 0. Этот случай обрабатывается отдельно. */
18      if (mid1.x == mid2.x) {
19        return new Point(mid1.x, mid1.y + ydir * size / 2.0);
20     }
21
22      double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
23      double x1 = 0;
24      double y1 = 0;
25
26      /* Наклон вычисляется по формуле (y1 - y2) / (x1 - x2).
```

```
27       * Примечание: при "крутом" наклоне (>1) конец отрезка
28       * пересечет горизонтальную сторону квадрата. При "пологом"
29       * наклоне (<1) конец отрезка пересечет вертикальную
30       * сторону квадрата. */
31      if (Math.abs(slope) == 1) {
32        x1 = mid1.x + xdir * size / 2.0;
33        y1 = mid1.y + ydir * size / 2.0;
34      } else if (Math.abs(slope) < 1) { // Пологий наклон
35        x1 = mid1.x + xdir * size / 2.0;
36        y1 = slope * (x1 - mid1.x) + mid1.y;
37      } else { // steep slope
38        y1 = mid1.y + ydir * size / 2.0;
39        x1 = (y1 - mid1.y) / slope + mid1.x;
40     }
41      return new Point(x1, y1);
42    }
43
44    public Line cut(Square other) {
45      /* Вычисление точек пересечения линии, соединяющей середины,
46       * со сторонами квадратов. */
47      Point p1 = extend(this.middle(), other.middle(), this.size);
48      Point p2 = extend(this.middle(), other.middle(), -1 * this.size);
49      Point p3 = extend(other.middle(), this.middle(), other.size);
50      Point p4 = extend(other.middle(), this.middle(), -1 * other.size);
51
52      /* Определение начала и конца отрезков. Начальная точка находится
53       * левее остальных (и выше при совпадении), а конечная - правее
54       * остальных (и ниже при совпадении). */
55      Point start = p1;
56      Point end = p1;
57      Point[] points = {p2, p3, p4};
58      for (int i = 0; i < points.length; i++) {
59        if (points[i].x < start.x ||
60            (points[i].x == start.x && points[i].y < start.y)) {
61          start = points[i];
62        } else if (points[i].x > end.x ||
63                (points[i].x == end.x && points[i].y > end.y)) {
64          end = points[i];
65      }
66     }
67
68      return new Line(start, end);
69    }
```

Основная цель этой задачи — увидеть, насколько вы внимательно относитесь к написанию кода. Очень легко забыть об особых случаях (когда середины квадратов совпадают по какой-либо из осей). Вы должны продумать поведение программы в особых ситуациях прежде, чем начнете писать код. Этот вопрос требует тщательного тестирования.

16.14. Для заданного набора точек на плоскости найдите линию, проходящую через максимальное количество точек.

РЕШЕНИЕ

На первый взгляд задача кажется довольно прямолинейной. И это ощущение правильно — в определенном отношении.

Просто проведем линию (не отрезок) между каждой парой точек и по хеш-таблице проверим, какая линия проходит через наибольшее количество точек. Это потребует времени $O(N^2)$, поскольку необходимо проанализировать N^2 пар.

Мы будем представлять линию величиной наклона и точкой пересечения оси y (вместо пары точек), потому что это позволит нам легко проверить эквивалентность линии, соединяющей точки `(x1, y1)` и `(x2, y2)`, с линией, соединяющей точки `(x3, y3)` и `(x4, y4)`.

Чтобы найти линию, соединяющую наибольшее количество точек, мы просто перебираем все линии и подсчитываем количество вхождений по хеш-таблице.

При этом существует одно маленькое затруднение. Две линии определяются равными, если они имеют одинаковый наклон и точку пересечения с осью y. По этой причине хеширование осуществляется по этим значениям (а конкретно по наклону). Проблема в том, что формат с плавающей точкой не всегда может быть точно представлен в двоичном виде. Для этого мы проверяем, находятся ли два вещественных числа на расстоянии менее `epsilon` друг от друга.

Что это означает для хеш-таблицы? То, что две линии с «равными» величинами наклона не могут хешироваться в одно значение. Для этого наклон округляется до ближайшего `epsilon`, а полученное значение `flooredSlope` используется как ключ хеширования. Затем для получения всех *потенциально* равных линий мы проводим поиск по хеш-таблице в трех точках: `flooredSlope`, `flooredSlope - epsilon` и `flooredSlope + epsilon`. Тем самым гарантируется, что были проверены все линии, которые могут оказаться равными.

```
1  /* Поиск линий, проходящих через наибольшее количество точек. */
2  Line findBestLine(GraphPoint[] points) {
3    HashMapList<Double, Line> linesBySlope = getListOfLines(points);
4    return getBestLine(linesBySlope);
5  }
6
7  /* Каждая пара точек добавляется в список как линия. */
8  HashMapList<Double, Line> getListOfLines(GraphPoint[] points) {
9    HashMapList<Double, Line> linesBySlope = new HashMapList<Double, Line>();
10   for (int i = 0; i < points.length; i++) {
11     for (int j = i + 1; j < points.length; j++) {
12       Line line = new Line(points[i], points[j]);
13       double key = Line.floorToNearestEpsilon(line.slope);
14       linesBySlope.put(key, line);
15     }
16   }
17   return linesBySlope;
18 }
19
20 /* Возвращает линию с наибольшим количеством эквивалентных линий. */
21 Line getBestLine(HashMapList<Double, Line> linesBySlope) {
22   Line bestLine = null;
23   int bestCount = 0;
24
```

```
25    Set<Double> slopes = linesBySlope.keySet();
26
27    for (double slope : slopes) {
28      ArrayList<Line> lines = linesBySlope.get(slope);
29      for (Line line : lines) {
30        /* Подсчет линий, эквивалентных текущей */
31        int count = countEquivalentLines(linesBySlope, line);
32
33        /* Если лучше текущей линии, заменить ее */
34        if (count > bestCount) {
35          bestLine = line;
36          bestCount = count;
37          bestLine.Print();
38          System.out.println(bestCount);
39      }
40     }
41    }
42    return bestLine;
43  }
44
45  /* Проверка эквивалентных линий по хеш-таблице. В проверку включается
46   * смещение +/- epsilon, так как две линии считаются эквивалентными,
47   * если они находятся в пределах epsilon друг от друга. */
48  int countEquivalentLines(HashMapList<Double, Line> linesBySlope, Line line) {
49    double key = Line.floorToNearestEpsilon(line.slope);
50    int count = countEquivalentLines(linesBySlope.get(key), line);
51    count += countEquivalentLines(linesBySlope.get(key - Line.epsilon), line);
52    count += countEquivalentLines(linesBySlope.get(key + Line.epsilon), line);
53    return count;
54  }
55
56  /* Подсчет в массиве линий, "эквивалентных" данной (у которых наклон
57   * и точка пересечения у отличаются не более чем на epsilon). */
58  int countEquivalentLines(ArrayList<Line> lines, Line line) {
59    if (lines == null) return 0;
60
61    int count = 0;
62    for (Line parallelLine : lines) {
63      if (parallelLine.isEquivalent(line)) {
64        count++;
65      }
66    }
67    return count;
68  }
69
70  public class Line {
71    public static double epsilon = .0001;
72    public double slope, intercept;
73    private boolean infinite_slope = false;
74
75    public Line(GraphPoint p, GraphPoint q) {
76      if (Math.abs(p.x - q.x) > epsilon) { // Если x различны
77        slope = (p.y - q.y) / (p.x - q.x); // вычислить наклон
78        intercept = p.y - slope * p.x;     // по формуле y=mx+b
79      } else {
80        infinite_slope = true;
81        intercept = p.x; // используется точка пересечения с x
```

```
82     }
83    }
84
85    public static double floorToNearestEpsilon(double d) {
86      int r = (int) (d / epsilon);
87      return ((double) r) * epsilon;
88    }
89
90    public boolean isEquivalent(double a, double b) {
91      return (Math.abs(a - b) < epsilon);
92    }
93
94    public boolean isEquivalent(Object o) {
95      Line l = (Line) o;
96       if (isEquivalent(l.slope, slope) && isEquivalent(l.intercept, intercept)
&&
97         (infinite_slope == l.infinite_slope)) {
98         return true;
99       }
100      return false;
101    }
102  }
103
104  /* HashMapList<String, Integer> связывает String с
105   * ArrayList<Integer>. Реализация приведена в приложении. */
```

Будьте внимательны с расчетом наклона линии. Линия может оказаться вертикальной, то есть она не пересекает ось y, а ее наклон бесконечен. Этот случай отслеживается с помощью флага (`infinite_slope`); он должен проверяться в методе `equals`.

16.15. В игру «Великий комбинатор» играют следующим образом:

В каждой из четырех ячеек находится шар красного (R), желтого (Y), зеленого (G) или синего (B) цвета. Последовательность RGGB означает, что в ячейке 1 — красный шар, в ячейках 2 и 3 — зеленый, 4 — синий. Пользователь должен угадать цвета шаров — например, YRGB. Если вы угадали правильный цвет в правильной ячейке, то вы получаете «хит». Если вы назвали цвет, который присутствует в раскладе, но находится в другой ячейке, вы получаете «псевдохит». Ячейка с «хитом» не засчитывается за «псевдохит». Например, если фактический расклад RGBY, а ваше предположение GGRR, то ответ должен быть: один «хит» и один «псевдохит».

Напишите метод, который возвращает число «хитов» и «псевдохитов».

РЕШЕНИЕ

Задача несложная, но даже в ней легко допустить второстепенные ошибки. *Тщательно* протестируйте свой код с учетом разных граничных случаев.

В нашей реализации будет использоваться массив с информацией о том, сколько раз каждый символ появлялся в `solution` (решении), исключая случаи, когда в ячейке «хит». Затем мы перебираем `guess` для подсчета количества псевдохитов.

Приведенный далее код реализует этот алгоритм:

```
class Result {
  public int hits = 0;
  public int pseudoHits = 0;

  public String toString() {
    return "(" + hits + ", " + pseudoHits + ")";
  }
}

int code(char c) {
  switch (c) {
  case 'B':
    return 0;
  case 'G':
    return 1;
  case 'R':
    return 2;
  case 'Y':
    return 3;
  default:
    return -1;
  }
}

int MAX_COLORS = 4;

Result estimate(String guess, String solution) {
  if (guess.length() != solution.length()) return null;

  Result res = new Result();
  int[] frequencies = new int[MAX_COLORS];

  /* Вычисление хитов и построение таблицы частот */
  for (int i = 0; i < guess.length(); i++) {
    if (guess.charAt(i) == solution.charAt(i)) {
      res.hits++;
    } else {
      /* Таблица частот (используемая для псевдохитов) увеличивается
       * только в том случае, если это не хит. */
      int code = code(solution.charAt(i));
      frequencies[code]++;
   }
  }

  /* Вычисление псевдохитов */
  for (int i = 0; i < guess.length(); i++) {
    int code = code(guess.charAt(i));
    if (code >= 0 && frequencies[code] > 0 &&
        guess.charAt(i) != solution.charAt(i)) {
      res.pseudoHits++;
      frequencies[code]--;
   }
  }
  return res;
}
```

Чем проще алгоритм, тем важнее на собеседовании написать чистый и правильный код. В данном случае мы выделили `code(char c)` в отдельный метод, а также создали класс `Result` для хранения результата вместо обычного его вывода.

16.16. Имеется массив целых чисел. Напишите метод для поиска таких индексов `m` и `n`, что для полной сортировки массива достаточно будет отсортировать элементы от `m` до `n`. Минимизируйте `n` и `m` (то есть найдите наименьшую такую последовательность).

Пример:

Ввод: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`

Вывод: `(3, 9)`

РЕШЕНИЕ

Прежде чем браться за решение, стоит разобраться в том, как должен выглядеть ответ. Если мы ищем пару индексов, это означает, что некоторая средняя часть массива нуждается в сортировке, а начало и конец массива уже упорядочены.

Рассмотрим пример:

```
1, 2, 4, 7, 10, 11, 8, 12, 5, 6, 16, 18, 19
```

Первая мысль — найти самую длинную возрастающую подпоследовательность в начале списка и самую длинную увеличивающуюся подпоследовательность в конце.

```
левая часть (left):     1, 2, 4, 7, 10, 11
средняя часть (middle): 8, 12
правая часть (right):   5, 6, 16, 18, 19
```

Сгенерировать такие подпоследовательности несложно: начинаем слева и справа и движемся внутрь. Когда встречается элемент, нарушающий порядок, это означает, что мы обнаружили конец увеличивающейся/уменьшающейся подпоследовательности.

Для решения задачи необходимо отсортировать среднюю часть массива, чтобы все элементы массива оказались упорядоченными. Таким образом, должны выполняться следующие условия:

```
/* все элементы слева меньше, чем все элементы в середине */
min(middle) > end(left)
/* все элементы средней части меньше всех элементов в правой */
max(middle) < start(right)
```

Или другими словами:

```
left < middle < right
```

Фактически, это условие *никогда* не будет соблюдаться. Середина по определению неупорядочена, то есть *всегда* `left.end > middle.start` и `middle.end > right.start`. То есть вы не сможете отсортировать середину, чтобы упорядочить весь массив.

Но мы можем *уменьшать* левые и правые подпоследовательности, пока исходные условия не станут соблюдаться. Левая часть должна быть меньше всех элементов в середине и правой части, а правая часть должна быть больше всех элементов в левой части и в середине.

Пусть `min = min(middle и right)` и `max = max (middle и left)`. Так как левая и правая части уже упорядочены, в действительности необходимо проверить только их начальную или конечную точку.

Слева мы начинаем двигаться от конца подпоследовательности (значение 11, элемент 5) и двигаемся влево. Значение `min` равно 5. Как только обнаруживается такой элемент `i`, что `array[i] < min`, это означает, что можно отсортировать середину, и часть массива будет упорядочена.

Затем то же самое делается справа. Значение `max` равно 12. Мы начинаем от начала правой подпоследовательности (значение 6) и перемещаемся вправо. Мы сравниваем `max` (12) с 6, затем с 7, затем с 16. При достижении 16 мы знаем, что не существует элементов, меньших 12 (увеличивающаяся подпоследовательность). Теперь можно отсортировать середину, чтобы весь массив стал упорядоченным.

Следующий код реализует этот алгоритм:

```
1  void findUnsortedSequence(int[] array) {
2    // Нахождение левой подпоследовательности
3    int end_left = findEndOfLeftSubsequence(array);
4    if (end_left >= array.length - 1) return; // Уже отсортировано
5
6    // Нахождение правой подпоследовательности
7    int start_right = findStartOfRightSubsequence(array);
8
9    // Определение минимума и максимума
10    int max_index = end_left; // Максимум левой стороны
11    int min_index = start_right; // Минимум правой стороны
12    for (int i = end_left + 1; i < start_right; i++) {
13      if (array[i] < array[min_index]) min_index = i;
14      if (array[i] > array[max_index]) max_index = i;
15    }
16
17    // Двигаться влево, пока не дойдем до array[min_index]
18    int left_index = shrinkLeft(array, min_index, end_left);
19
20    // Двигаться вправо, пока не дойдем до array[max_index]
21    int right_index = shrinkRight(array, max_index, start_right);
22
23    System.out.println(left_index + " " + right_index);
24  }
25
26  int findEndOfLeftSubsequence(int[] array) {
27    for (int i = 1; i < array.length; i++) {
28      if (array[i] < array[i - 1]) return i - 1;
29    }
30    return array.length - 1;
31  }
32
33  int findStartOfRightSubsequence(int[] array) {
34    for (int i = array.length - 2; i >= 0; i--) {
35      if (array[i] > array[i + 1]) return i + 1;
36    }
37    return 0;
38  }
39
40  int shrinkLeft(int[] array, int min_index, int start) {
```

```
41    int comp = array[min_index];
42    for (int i = start - 1; i >= 0; i--) {
43      if (array[i] <= comp) return i + 1;
44    }
45    return 0;
46  }
47
48  int shrinkRight(int[] array, int max_index, int start) {
49    int comp = array[max_index];
50    for (int i = start; i < array.length; i++) {
51      if (array[i] >= comp) return i - 1;
52    }
53    return array.length - 1;
54  }
```

Обратите внимание на использование других методов в этом решении. Можно было бы сгруппировать весь код в одном методе, но это существенно усложнит чтение, сопровождение и тестирование кода. На собеседовании нужно уделять первостепенное внимание этим аспектам.

16.17. Имеется массив целых чисел (как положительных, так и отрицательных). Найдите непрерывную последовательность с наибольшей суммой. Верните найденную сумму.

Пример:
Ввод: `2, -8, 3, -2, 4, -10`
Вывод: `5 (соответствующая последовательность: 3, -2, 4)`

РЕШЕНИЕ

Эта довольно сложная задача часто встречается на собеседованиях. Начнем с примера:

```
2   3   -8   -1   2   4   -2   3
```

Если рассматривать массив как содержащий чередующиеся последовательности положительных и отрицательных чисел, то не имеет смысла рассматривать отдельные части положительных или отрицательных подпоследовательностей. Почему? Включение части отрицательной подпоследовательности уменьшит итоговое значение суммы, а значит, лучше было бы не включать отрицательную подпоследовательность вообще. Включение части положительной подпоследовательности выглядит еще более странным, поскольку включение этой подпоследовательности целиком всегда даст больший результат.

Нужно придумать алгоритм, рассматривая массив как последовательность чередующихся отрицательных и положительных чисел. Каждое число соответствует сумме подпоследовательности положительных и отрицательных чисел. В нашем примере массив можно сократить до:

```
5   -9   6   -2   3
```

Мы еще не получили итоговый алгоритм, но зато стало более понятно, с чем мы имеем дело.

Рассмотрим предыдущий массив. Нужно ли учитывать подпоследовательность {5, -9}? Нет. Эти числа в сумме дают –4, а значит, нет смысла учитывать оба этих числа, достаточно только {5}.

В каких случаях имеет смысл учитывать отрицательные числа? Только если это позволяет нам объединить две положительные подпоследовательности, сумма каждой из которых больше, чем вклад отрицательной величины.

Эту процедуру лучше всего реализовать поэтапно, начиная с первого элемента в массиве.

5 — это самая большая сумма, встретившаяся нам. Таким образом, maxsum=5 и sum=5. Далее рассматривается следующее число (–9). Если прибавить это число к sum, то получится отрицательная величина. Нет смысла расширять подпоследовательность с 5 до –9 (–9 уменьшает общую сумму до –4). Таким образом, мы просто сбрасываем значение sum.

Теперь рассматривается следующий элемент (6). Эта подпоследовательность больше 5, поэтому мы обновляем значения maxsum и sum.

Затем рассматривается следующий элемент (–2). Прибавление этого числа к 6 дает sum=4. Так как эта «добавка» может быть полезной (при добавлении к другой, большей последовательности), возможно, {6, -2} *желательно* включить в максимальную подпоследовательность. Мы обновляем sum, но не maxsum.

Наконец, рассматривается следующий элемент (3). Прибавление 3 к sum (4) дает 7, поэтому мы обновляем maxsum. Максимальная последовательность имеет вид {6, -2, 3}.

При работе с развернутым массивом логика остается такой же. Ниже приведена реализация этого алгоритма:

```
1  int getMaxSum(int[] a) {
2    int maxsum = 0;
3    int sum = 0;
4    for (int i = 0; i < a.length; i++) {
5      sum += a[i];
6      if (maxsum < sum) {
7        maxsum = sum;
8      } else if (sum < 0) {
9        sum = 0;
10     }
11   }
12   return maxsum;
13 }
```

А если массив состоит из одних отрицательных чисел? Как действовать в этом случае? Возьмем простой массив {-3, -10, -5}. Можно дать три разных ответа:

- –3 (если считать, что субпоследовательность не может быть пустой);
- 0 (субпоследовательность может иметь нулевую длину);
- MINIMUM_INT (признак ошибки).

В нашем коде был использован второй вариант (sum=0), но в этом вопросе не существует однозначного «правильного» решения. Обсудите этот вопрос с интервьюером; там самым вы продемонстрируете свое внимание к мелочам.

16.18. Заданы две строки — pattern и value. Строка pattern состоит из букв a и b, описывающих шаблон чередования подстрок в строке. Например, строка catcatgocatgo соответствует шаблону aabab (где cat представляет a, а go — b). Также строка соответствует таким шаблонам, как a, ab и b.

Напишите метод для определения того, соответствует ли value шаблону pattern.

РЕШЕНИЕ

Как обычно, начнем с решения методом «грубой силы».

Метод «грубой силы»

Примитивный алгоритм просто пытается перебрать все возможные значения a и b, а затем проверяет, что из этого получилось.

Для этого мы можем перебрать все возможные подстроки a и все возможные подстроки b. Строка длиной n имеет $O(n^2)$ подстрок, поэтому операция фактически будет выполняться за время $O(n^4)$. Но для каждого значения a и b необходимо построить новую строку соответствующей длины и проверить ее на равенство. Шаг построения/сравнения выполняется за время $O(n)$, и в результате общее время выполнения составит $O(n^5)$.

```
1  Для каждой возможной подстроки a
2    Для каждой возможной подстроки b
3      candidate = buildFromPattern(pattern, a, b)
4      Если candidate == value
5      Вернуть true
```

Одна простая оптимизация основана на том факте, что если pattern начинается с a, то и в начале value должна находиться строка a (в противном случае в начале value должна находиться строка b). Следовательно, количество возможных значений a сокращается с $O(n^2)$ до $O(n)$.

В этом случае алгоритм проверяет, начинается ли шаблон pattern с a или b. Если шаблон начинается с b, то его можно «инвертировать» (преобразовать все a в b, а все b в a), чтобы он начинался с a. Далее перебираются все возможные подстроки для a (каждая из которых должна начинаться в позиции 0) и все возможные подстроки для b (каждая из которых должна начинаться с некоторого символа за концом a). Как и прежде, затем строка, соответствующая шаблону, сравнивается с предыдущей строкой. Улучшенная версия выполняется за время $O(n^4)$.

Существует еще одна второстепенная оптимизация: если строка начинается с b вместо a, выполнять «инверсию» не обязательно — этот факт можно учесть в методе buildFromPattern. Первый символ pattern рассматривается как «основной», а другой — как «альтернативный». Метод buildFromPattern строит подходящую строку в зависимости от того, каким символом является a — основным или альтернативным.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    int size = value.length();
5    for (int mainSize = 0; mainSize < size; mainSize++) {
```

```
6      String main = value.substring(0, mainSize);
7      for (int altStart = mainSize; altStart <= size; altStart++) {
8        for (int altEnd = altStart; altEnd <= size; altEnd++) {
9          String alt = value.substring(altStart, altEnd);
10          String cand = buildFromPattern(pattern, main, alt);
11        if (cand.equals(value)) {
12            return true;
13        }
14       }
15      }
16    }
17    return false;
18  }
19
20  String buildFromPattern(String pattern, String main, String alt) {
21    StringBuffer sb = new StringBuffer();
22    char first = pattern.charAt(0);
23    for (char c : pattern.toCharArray()) {
24      if (c == first) {
25        sb.append(main);
26      } else {
27        sb.append(alt);
28      }
29    }
30    return sb.toString();
31  }
```

Конечно, нужно искать более эффективный алгоритм.

Оптимизированный алгоритм

Поразмыслим над текущей версией алгоритма. Поиск по всем значениям основной строки выполняется достаточно быстро (за время $O(n)$); медленно работает альтернативная строка: $O(n^2)$. Как оптимизировать этот процесс?

Допустим, шаблон вида `aabab` сравнивается со строкой `catcatgocatgo`. После того как мы выберем строку `cat` для проверки, строки `a` займут до 9 символов (3 строки по 3 символа каждая). Следовательно, строки `b` должны занимать оставшиеся 4 символа, и каждая из них должна иметь длину 2. Кроме того, мы точно знаем, где они должны находиться: если `a` представляет строку `cat`, а строка `pattern` содержит символы `aabab`, то `b` должны соответствовать символы `go`.

Иначе говоря, выбор `a` также определяет выбор `b`. Перебор не нужен — общей статистики по шаблону (количество `a`, количество `b`, первое вхождение каждого подшаблона) и перебора значений `a` (а вернее, основной строки) будет достаточно.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
10   int firstAlt = pattern.indexOf(altChar);
```

```
11    int maxMainSize = size / countOfMain;
12
13    for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14      int remainingLength = size - mainSize * countOfMain;
15      String first = value.substring(0, mainSize);
16      if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
17        int altIndex = firstAlt * mainSize;
18        int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
19        String second = countOfAlt == 0 ? "" :
20                        value.substring(altIndex, altSize + altIndex);
21
22        String cand = buildFromPattern(pattern, first, second);
23        if (cand.equals(value)) {
24          return true;
25      }
26     }
27    }
28    return false;
29  }
30
31  int countOf(String pattern, char c) {
32    int count = 0;
33    for (int i = 0; i < pattern.length(); i++) {
34      if (pattern.charAt(i) == c) {
35        count++;
36     }
37    }
38    return count;
39  }
40
41  String buildFromPattern(...) {  /* См. выше */ }
```

Алгоритм выполняется за время $O(n^2)$, потому что мы перебираем $O(n)$ возможностей для основной строки и выполняем работу $O(n)$ для построения и сравнения строк.

Заметим, что оптимизация также сокращает количество возможностей для основной строки. Если основная строка существует в трех экземплярах, то ее длина не может быть больше трети длины `value`.

Оптимизация (альтернативная)

Если вам не нравится идея создавать строку только для ее сравнения (а затем уничтожать ее), этот шаг можно исключить.

Вместо этого можно перебирать значения `a` и `b`, как и прежде. Но на этот раз для проверки соответствия строки шаблону (для заданных значений `a` и `b`) мы перебираем `value` и сравниваем каждую подстроку с первым вхождением строк `a` и `b`.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
```

```
10    int firstAlt = pattern.indexOf(altChar);
11    int maxMainSize = size / countOfMain;
12
13    for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14      int remainingLength = size - mainSize * countOfMain;
15      if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
16        int altIndex = firstAlt * mainSize;
17        int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
18        if (matches(pattern, value, mainSize, altSize, altIndex)) {
19          return true;
20      }
21     }
22    }
23    return false;
24  }
25
26  /* Перебор по шаблону и значению. Для каждого символа в шаблоне
27   * проверяем, является ли он основным или альтернативным. Затем
28   * проверяем, соответствует ли следующая группа символов value
29   * исходному набору символов (основному или альтернативному). */
30  boolean matches(String pattern, String value, int mainSize, int altSize,
31                  int firstAlt) {
32    int stringIndex = mainSize;
33    for (int i = 1; i < pattern.length(); i++) {
34      int size = pattern.charAt(i) == pattern.charAt(0) ? mainSize : altSize;
35      int offset = pattern.charAt(i) == pattern.charAt(0) ? 0 : firstAlt;
36      if (!isEqual(value, offset, stringIndex, size)) {
37        return false;
38     }
39      stringIndex += size;
40    }
41    return true;
42  }
43
44  /* Проверка равенства двух подстрок с заданными смещениями
45   * и размером size. */
46  boolean isEqual(String s1, int offset1, int offset2, int size) {
47    for (int i = 0; i < size; i++) {
48      if (s1.charAt(offset1 + i) != s1.charAt(offset2 + i)) {
49        return false;
50     }
51    }
52    return true;
53  }
```

Алгоритм выполняется за время $O(n^2)$, но его преимуществом является ускоренное прекращение проверки при раннем несовпадении (как это обычно происходит). Предыдущий алгоритм узнает о неудаче только после выполнения всей работы по построению строки.

16.19. Целочисленная матрица представляет участок земли; значение каждого элемента матрицы обозначает высоту над уровнем моря. Нулевые элементы представляют воду. Назовем «озером» область водных участков, связанных по вертикали, горизонтали или диагонали. Размер озера равен общему количеству соединенных водных участков. Напишите метод для вычисления размеров всех озер в матрице.

Пример:
Ввод:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Вывод: 2, 4, 1 *(в любом порядке)*

РЕШЕНИЕ

Для начала опробуем простой перебор массива. Найти «водные» ячейки легко: они содержат 0.

Как для известной ячейки с водой вычислить количество воды в окрестностях? Если по соседству нет других ячеек, содержащих нули, то размер озера равен 1. Если такие ячейки существуют, нужно добавить соседние ячейки, а также все водные ячейки, соседние с ними. Конечно, нужно проследить за тем, чтобы ячейки не учитывались повторно. Для этого можно воспользоваться модифицированным вариантом поиска в глубину или ширину с пометкой посещенных ячеек.

Для каждой ячейки необходимо проверить восемь соседних ячеек. Для этого можно написать отдельные строки для проверки верхней, нижней, правой, левой и каждой из четырех диагональных ячеек. Впрочем, будет проще воспользоваться циклом `for`.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2    ArrayList<Integer> pondSizes = new ArrayList<Integer>();
3    for (int r = 0; r < land.length; r++) {
4      for (int c = 0; c < land[r].length; c++) {
5        if (land[r][c] == 0) { // Проверка не обязательна
6          int size = computeSize(land, r, c);
7          pondSizes.add(size);
8      }
9     }
10    }
11    return pondSizes;
12  }
13
14  int computeSize(int[][] land, int row, int col) {
15    /* Если ячейка выходит за границы участка или уже посещалась */
16    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
17        land[row][col] != 0) { // Посещалась или не содержит воду
18      return 0;
19    }
20    int size = 1;
21    land[row][col] = -1; // Ячейка помечается как посещенная
22    for (int dr = -1; dr <= 1; dr++) {
23      for (int dc = -1; dc <= 1; dc++) {
24        size += computeSize(land, row + dr, col + dc);
25      }
26    }
27    return size;
28  }
```

В этом примере для пометки посещенных ячеек им присваивается значение –1. Это позволяет нам проверить в одном условии (`land[row][col] != 0`), содержит ли ячейка землю или была посещена ранее.

Возможно, вы также заметили, что в цикле `for` перебираются девять ячеек вместо восьми, то есть в перебор включается текущая ячейка. Можно было бы добавить строку, которая отменяет рекурсию для `dr == 0` и `dc == 0`. На самом деле никакой экономии это не даст: мы будем выполнять лишнюю команду `if` для восьми ячеек только для того, чтобы предотвратить один рекурсивный вызов. Рекурсивный вызов немедленно вернет управление, так как ячейка помечена как посещенная.

Если вам не хочется изменять входную матрицу, создайте вспомогательную матрицу `visited`.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2    boolean[][] visited = new boolean[land.length][land[0].length];
3    ArrayList<Integer> pondSizes = new ArrayList<Integer>();
4    for (int r = 0; r < land.length; r++) {
5      for (int c = 0; c < land[r].length; c++) {
6        int size = computeSize(land, visited, r, c);
7        if (size > 0) {
8          pondSizes.add(size);
9      }
10     }
11   }
12   return pondSizes;
13 }
14
15 int computeSize(int[][] land, boolean[][] visited, int row, int col) {
16   /* Если ячейка выходит за границы участка или уже посещалась */
17   if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
18       visited[row][col] || land[row][col] != 0) {
19     return 0;
20   }
21   int size = 1;
22   visited[row][col] = true;
23   for (int dr = -1; dr <= 1; dr++) {
24     for (int dc = -1; dc <= 1; dc++) {
25       size += computeSize(land, visited, row + dr, col + dc);
26    }
27   }
28   return size;
29 }
```

Обе реализации имеют сложность $O(WH)$, где W — ширина, а H — высота матрицы.

Примечание: многие кандидаты говорят «$O(N)$» или «$O(N2)$», словно в обозначение N заложен некий изначальный смысл. Тем не менее это не так. Представьте, что матрица квадратная; тогда время выполнения может описываться обозначением $O(N)$ или $O(N2)$. Оба обозначения верны в зависимости от того, что понимается под N: количество ячеек или длина стороны.

Иногда на собеседованиях встречается неправильная оценка времени выполнения $O(N^4)$. Предполагается, что метод `computeSize` выполняется за время $O(N2)$, и он может вызываться до $O(N^2)$ раз (предполагается матрица размером $N \times N$). Хотя оба утверждения правильны, просто перемножать их нельзя, потому что с повышением затрат на вызов `computeSize` количество его вызовов уменьшается.

Допустим, что первый вызов `computeSize` обходит всю матрицу. Это может потребовать времени $O(N^2)$, но после этого `computeSize` вызываться не будет.

Для получения указанной оценки также можно подумать над тем, сколько раз к каждой ячейке обращается каждый вызов. Функция `computePondSizes` один раз обращается к каждой ячейке. Кроме того, к ячейке может однократно обратиться каждая из ее соседних ячеек. Число обращений к каждой ячейке остается постоянным. Следовательно, общее время выполнения составит $O(N^2)$ для матрицы $N \times N$, или в более общем виде — $O(WH)$.

16.20. На старых сотовых телефонах при наборе текста пользователь нажимал клавиши на цифровой клавиатуре, а телефон выдавал список слов, соответствующих введенным цифрам. Каждая цифра представляла набор от 0 до 4 букв. Реализуйте алгоритм для получения списка слов, соответствующих заданной последовательности цифр. Предполагается, что список допустимых слов задан (в любой структуре данных на ваш выбор). Следующая диаграмма поясняет процесс подбора:

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
	0	

Пример:

Ввод: `8733`

Вывод: `tree, used`

РЕШЕНИЕ

Начнем с решения методом «грубой силы».

Метод «грубой силы»

Как бы вы стали решать эту задачу, если бы вам пришлось делать это вручную? Вероятно, вы бы опробовали каждое возможное значение для каждой цифры со всеми остальными возможными значениями.

Именно так будет работать наш алгоритм. Мы возьмем первую цифру и переберем все символы, соответствующие этой цифре. Каждый символ добавляется к переменной `prefix`, и происходит рекурсия с дальнейшей передачей `prefix`.

Когда все символы будут обработаны, алгоритм выводит значение переменной `prefix` (которая теперь содержит полное слово), если строка содержит действительное слово.

Будем считать, что список слов передается в виде `HashSet`. Эта коллекция похожа на хеш-таблицу, но вместо поиска по ключу она может проверить, содержится ли слово в множестве, за время $O(1)$.

```
ArrayList<String> getValidT9Words(String number, HashSet<String> wordList) {
  ArrayList<String> results = new ArrayList<String>();
  getValidWords(number, 0, "", wordList, results);
  return results;
}

void getValidWords(String number, int index, String prefix,
                   HashSet<String> wordSet, ArrayList<String> results) {
  /* Если получилось законченное слово, вывести его. */
  if (index == number.length() && wordSet.contains(prefix)) {
    results.add(prefix);
    return;
  }

  /* Получение символов, соответствующих данной цифре. */
  char digit = number.charAt(index);
  char[] letters = getT9Chars(digit);

  /* Перебор всех оставшихся вариантов. */
  if (letters != null) {
    for (char letter : letters) {
      getValidWords(number, index + 1, prefix + letter, wordSet, results);
    }
  }
}

/* Получение массива символов, соответствующих цифре. */
char[] getT9Chars(char digit) {
  if (!Character.isDigit(digit)) {
    return null;
  }
  int dig = Character.getNumericValue(digit) - Character.getNumericValue('0');
  return t9Letters[dig];
}

/* Отображение цифр на буквы. */
char[][] t9Letters = {null, null, {'a', 'b', 'c'}, {'d', 'e', 'f'},
  {'g', 'h', 'i'}, {'j', 'k', 'l'}, {'m', 'n', 'o'}, {'p', 'q', 'r', 's'},
  {'t', 'u', 'v'},  {'w', 'x', 'y', 'z'}
};
```

Алгоритм выполняется за время $O(4^N)$, где N — длина строки. Это объясняется тем, что для каждого вызова `getValidWords` выполняется 4-кратное рекурсивное ветвление, а рекурсия продолжается до уровня N. Для больших строк такое решение работает очень, очень медленно.

Оптимизированное решение

Вернемся к тому, как бы вы действовали при ручном решении этой задачи. Возьмем пример 33835676368 (соответствующий слову «development».) Если бы вы решали эту задачу вручную, наверняка вы бы пропустили решения, начинающиеся с fftf [3383], так как не существует ни одного слова, начинающегося с этих символов.

В идеале наша программа должна применять аналогичную оптимизацию: прекращать рекурсию по тому пути, который очевидно приведет к неудаче. А именно рекурсия прерывается, если в словаре нет ни одного слова, начинающегося с `prefix`.

Структура данных нагруженного дерева (с. 97) поможет нам в решении этой задачи. Каждый раз при достижении строки, не являющейся действительным префиксом, рекурсия прерывается.

```
1  ArrayList<String> getValidT9Words(String number, Trie trie) {
2    ArrayList<String> results = new ArrayList<String>();
3    getValidWords(number, 0, "", trie.getRoot(), results);
4    return results;
5  }
6
7  void getValidWords(String number, int index, String prefix, TrieNode trieNode,
8                     ArrayList<String> results) {
9    /* Если получилось законченное слово, вывести его. */
10    if (index == number.length()) {
11      if (trieNode.terminates()) { // Является ли полным словом?
12        results.add(prefix);
13     }
14      return;
15    }
16
17    /* Получение символов, соответствующих данной цифре */
18    char digit = number.charAt(index);
19    char[] letters = getT9Chars(digit);
20
21    /* Перебор остальных вариантов. */
22    if (letters != null) {
23      for (char letter : letters) {
24        TrieNode child = trieNode.getChild(letter);
25        /* Если существуют слова, начинающиеся с prefix+letter,
26         * продолжить рекурсию. */
27        if (child != null) {
28          getValidWords(number, index + 1, prefix + letter, child, results);
29      }
30     }
31    }
32  }
```

Оценить время выполнения этого алгоритма достаточно сложно, потому что оно зависит от конкретного языка. Тем не менее эта оптимизация намного ускорит выполнение кода на практике.

Оптимальное решение

Хотите верьте, хотите нет, но алгоритм может работать еще быстрее — нужно лишь провести небольшую предварительную обработку. Впрочем, это не создаст существенных проблем, потому что она все равно понадобится для построения нагруженного дерева.

В задаче требуется вернуть список всех слов, соответствующих заданной последовательности цифр в T9. Вместо того чтобы пытаться делать это «на ходу» (с перебором большого количества вариантов, многие из которых работать не будут), можно выполнить всю работу заранее.

Алгоритм состоит из нескольких шагов:

1. Создать хеш-таблицу, связывающую последовательность цифр со списком строк.

2. Перебрать все слова в словаре и преобразовать их в эквивалентные последовательности T9 (например, APPLE -> 27753). Сохранить эти отображения в хеш-таблице. Например, 8733 отображается на пару {used, tree}.
3. Провести в хеш-таблице выборку по ключу и вернуть список.

Вот и все!

```
1  /* Выборка слов по ключу */
2  ArrayList<String> getValidT9Words(String numbers,
3                                    HashMapList<String, String> dictionary) {
4    return dictionary.get(numbers);
5  }
6
7  /* ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА */
8
9  /* Построение хеш-таблицы, связывающей последовательность цифр
10   *  со всеми словами, имеющими такое представление. */
11 HashMapList<String, String> initializeDictionary(String[] words) {
12   /* Построение хеш-таблицы, связывающей буквы с цифрами */
13   HashMap<Character, Character> letterToNumberMap = createLetterToNumberMap();
14
15   /* Создание отображения "слово -> последовательность цифр". */
16   HashMapList<String, String> wordsToNumbers = new HashMapList<String,
String>();
17   for (String word : words) {
18     String numbers = convertToT9(word, letterToNumberMap);
19     wordsToNumbers.put(numbers, word);
20   }
21   return wordsToNumbers;
22 }
23
24 /* Переход к отображению "буква -> цифра". */
25 HashMap<Character, Character> createLetterToNumberMap() {
26   HashMap<Character, Character> letterToNumberMap =
27     new HashMap<Character, Character>();
28   for (int i = 0; i < t9Letters.length; i++) {
29     char[] letters = t9Letters[i];
30     if (letters != null) {
31       for (char letter : letters) {
32         char c = Character.forDigit(i, 10);
33         letterToNumberMap.put(letter, c);
34       }
35     }
36   }
37   return letterToNumberMap;
38 }
39
40 /* Преобразование строки в последовательность цифр T9. */
41 String convertToT9(String word, HashMap<Character, Character>
letterToNumberMap) {
42   StringBuilder sb = new StringBuilder();
43   for (char c : word.toCharArray()) {
44     if (letterToNumberMap.containsKey(c)) {
45       char digit = letterToNumberMap.get(c);
46       sb.append(digit);
47     }
48   }
```

```
49    return sb.toString();
50  }
51
52  char[][] t9Letters = /* См. выше */
53
54  /* HashMapList<String, Integer> связывает String
55   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Получение слов, соответствующих заданной последовательности цифр, выполняется за время $O(N)$, где N — количество цифр. Сложность $O(N)$ происходит от выборки из хеш-таблицы (последовательность цифр преобразуется в данные). Если известно, что длина слов не превышает некоторого максимума, время выполнения также можно описать как $O(1)$.

Возможно, кто-то подумает: «Линейное время — не так уж и быстро». Но все зависит от того, какой фактор создает линейную зависимость. Линейная зависимость от длины слова работает в высшей степени быстро, а линейная зависимость от длины словаря — намного медленнее.

16.21. Имеются два целочисленных массива. Найдите пару значений (по одному из каждого массива), которые можно поменять местами, чтобы суммы элементов двух массивов были одинаковыми.

Пример:

Ввод: `{4, 1, 2, 1, 1, 2}` и `{3, 6, 3, 3}`

Вывод: `{1, 3}`

РЕШЕНИЕ

Для начала нужно понять, что же именно требуется найти в этой задаче.

Даны два массива с суммами. Скорее всего, суммы заранее не известны, но мы пока будем действовать так, будто знаем их. В конце концов, вычисление суммы является операцией со сложностью $O(N)$, а мы знаем, что сложность $O(N)$ превзойти все равно не удастся. Следовательно, вычисление суммы не отразится на времени выполнения.

При перемещении (положительного) значения `a` из массива `A` в массив `B` сумма `A` уменьшается на `a`, а сумма `B` увеличивается на `a`. А значит, мы ищем два значения `a` и `b`, для которых:

```
sumA - a + b = sumB - b + a
```

После простых математических преобразований:

```
2a - 2b = sumA - sumB
a - b = (sumA - sumB) / 2
```

Следовательно, мы ищем два значения с конкретной целевой разностью: `(sumA--sumB) / 2`.

Заметим, что поскольку целевая разность должна быть целым числом (в конце концов, невозможно переставить два целых числа для получения нецелой разности), из этого следует, что для существования искомой пары разность между суммами должна быть четной.

Метод «грубой силы»

Алгоритм «грубой силы» достаточно прост: нужно перебрать содержимое массивов и проверить все пары значений. Это делается либо «наивным» способом (сравнением новых сумм), либо поиском пары с целевой разностью.

«Наивное» решение:

```
int[] findSwapValues(int[] array1, int[] array2) {
  int sum1 = sum(array1);
  int sum2 = sum(array2);

  for (int one : array1) {
    for (int two : array2) {
      int newSum1 = sum1 - one + two;
      int newSum2 = sum2 - two + one;
      if (newSum1 == newSum2) {
         int[] values = {one, two};
         return values;
      }
    }
  }

  return null;
}
```

Решение с целевой разностью:

```
int[] findSwapValues(int[] array1, int[] array2) {
  Integer target = getTarget(array1, array2);
  if (target == null) return null;

  for (int one : array1) {
    for (int two : array2) {
      if (one - two == target) {
        int[] values = {one, two};
        return values;
      }
    }
  }

  return null;
}

Integer getTarget(int[] array1, int[] array2) {
  int sum1 = sum(array1);
  int sum2 = sum(array2);

  if ((sum1 - sum2) % 2 != 0) return null;
  return (sum1 - sum2) / 2;
}
```

Для возвращаемого значения `getTarget` используется «упакованный» (boxed) тип данных `Integer`, чтобы мы могли отличить признак ошибки.

Алгоритм выполняется за время $O(AB)$.

Оптимальное решение

Задача сводится к поиску пары значений с конкретной разностью. Учитывая этот факт, вернемся к тому, что происходит в решении методом «грубой силы».

В этом решении мы перебираем содержимое `A`, и для каждого элемента ищем в `B` элемент, который дает «правильную» разность. Если значение в `A` равно 5, а целевое значение равно 3, то искать следует значение 2. Только с ним может быть достигнута поставленная цель.

Таким образом, вместо условия `one - two == target` с таким же успехом можно проверять условие `two == one - target`.

Насколько быстро мы сможем найти в `B` элемент, равный `one - target`?

При помощи хеш-таблицы это делается очень быстро. Достаточно поместить все элементы `B` в хеш-таблицу, затем перебрать `A` и провести поиск соответствующего элемента в `B`.

```
1  int[] findSwapValues(int[] array1, int[] array2) {
2    Integer target = getTarget(array1, array2);
3    if (target == null) return null;
4    return findDifference(array1, array2, target);
5  }
6
7  /* Поиск пары значений с конкретной разностью. */
8  int[] findDifference(int[] array1, int[] array2, int target) {
9    HashSet<Integer> contents2 = getContents(array2);
10   for (int one : array1) {
11     int two = one - target;
12     if (contents2.contains(two)) {
13       int[] values = {one, two};
14       return values;
15     }
16   }
17
18   return null;
19 }
20
21 /* Включение содержимого массива в HashSet. */
22 HashSet<Integer> getContents(int[] array) {
23   HashSet<Integer> set = new HashSet<Integer>();
24   for (int a : array) {
25     set.add(a);
26   }
27   return set;
28 }
```

Это решение выполняется за время $O(A + B)$. Это время является лучшим из возможных, так как нам придется как минимум обратиться к каждому элементу обоих массивов.

Альтернативное решение

Если массивы отсортированы, мы можем перебрать их для поиска подходящей пары. Это решение сократит затраты памяти.

```
int[] findSwapValues(int[] array1, int[] array2) {
  Integer target = getTarget(array1, array2);
  if (target == null) return null;
  return findDifference(array1, array2, target);
}

int[] findDifference(int[] array1, int[] array2, int target) {
  int a = 0;
  int b = 0;

   while (a < array1.length && b < array2.length) {
     int difference = array1[a] - array2[b];
     /* Сравнить difference с target. Если разность difference слишком
      * мала, увеличить ее, переместив a к большему значению. Если она
      * слишком велика, уменьшить ее, переместив b к большему значению.
      * Если найдена искомая величина, вернуть пару. */
     if (difference == target) {
       int[] values = {array1[a], array2[b]};
       return values;
     } else if (difference < target) {
       a++;
     } else {
       b++;
    }
   }

   return null;
}
```

Этот алгоритм выполняется за время $O(A + B)$, но требует, чтобы массивы были отсортированы. Если массивы не отсортированы, то алгоритм все равно можно применить, но массивы придется сначала отсортировать. Общее время выполнения составит $O(A \log A + B \log B)$.

16.22. Муравей сидит на бесконечной сетке из белых и черных квадратов. В исходном состоянии он смотрит вправо. На каждом шаге он поступает следующим образом:

(1) На белом квадрате муравей изменяет цвет квадрата, поворачивает на 90 градусов направо (по часовой стрелке) и перемещается вперед на один квадрат.

(2) На черном квадрате муравей изменяет цвет квадрата, поворачивает на 90 градусов налево (против часовой стрелки) и перемещается вперед на один квадрат.

Напишите программу, моделирующую первые K перемещений муравья. Выведите итоговое состояние сетки. Структура данных для представления сетки не задана — вы должны спроектировать ее самостоятельно. Входные данные метода состоят из значения K. Выведите итоговое состояние сетки без возвращения результата. Сигнатура метода должна выглядеть примерно так: `void printKMoves(int K)`.

РЕШЕНИЕ

На первый взгляд задача кажется очень прямолинейной: создать сетку, запомнить позицию и ориентацию муравья, изменить цвет ячейки, повернуть и переместиться. Интересная часть задачи связана с представлением бесконечной сетки.

Решение 1. Массив фиксированного размера

Формально в задаче моделируются только первые K перемещений, поэтому размер сетки ограничен. Муравей не может переместиться более чем на K ячеек в любом направлении. Если создать сетку с шириной 2K и высотой 2K (и посадить муравья в центре), этого будет заведомо достаточно.

К сожалению, это решение плохо расширяется. Если после K перемещений вы захотите выполнить еще K перемещений, могут возникнуть проблемы.

Кроме того, в таком решении большая часть пространства расходуется неэффективно. Возможно, максимальное смещение составит K ячеек в каком-то направлении, но может оказаться, что какое-то расстояние будет пройдено по кругу. Вполне возможно, что не все пространство будет реально использоваться.

Решение 2. Массив переменного размера

Возникает естественное желание использовать массив переменного размера, например класс Java `ArrayList`. Это позволит расширять массив по мере необходимости, сохраняя амортизированное время вставки $O(1)$.

Проблема в том, что сетка должна расширяться в двух измерениях, но `ArrayList` представляет одномерный массив. Кроме того, необходимо предусмотреть расширение в отрицательном, то есть обратном направлении. Класс `ArrayList` такой возможности не поддерживает.

Впрочем, мы можем пойти по аналогичному пути и построить собственную сетку с переменным размером. Каждый раз, когда муравей добирается до края, мы удваиваем размер сетки в этом направлении.

А как насчет отрицательного расширения? Хотя концептуально можно рассматривать элементы, находящиеся в отрицательных позициях, обратиться к элементу массива с отрицательным индексом не удастся.

Одно из возможных решений основано на создании «фиктивных индексов». Например, мы считаем, что муравей находится в ячейке с координатами $(-3, -10)$, но при этом вводим смещения для преобразования этих координат в реальные индексы массива.

Впрочем, на самом деле это не нужно. Позиция муравья не обязана быть общедоступной или логически непротиворечивой (если, конечно, такое требование не было выдвинуто интервьюером). Когда муравей перемещается в область отрицательных координат, мы можем удвоить размер массива и переместить муравья вместе со всеми ячейками в область положительных координат. Фактически происходит переназначение индексов. Оно не влияет на сложность в «O-записи», так как в любом случае приходится создавать новую матрицу.

```
1  public class Grid {
2    private boolean[][] grid;
```

```
3    private Ant ant = new Ant();
4
5    public Grid() {
6      grid = new boolean[1][1];
7    }
8
9    /* Скопировать старые значения в новый массив, с применением
10    * смещений к строкам/столбцам. */
11   private void copyWithShift(boolean[][] oldGrid, boolean[][] newGrid,
12                              int shiftRow, int shiftColumn) {
13     for (int r = 0; r < oldGrid.length; r++) {
14       for (int c = 0; c < oldGrid[0].length; c++) {
15         newGrid[r + shiftRow][c + shiftColumn] = oldGrid[r][c];
16       }
17     }
18   }
19
20   /* Убедиться в том, что заданная позиция помещается в массиве.
21    * При необходимости удвоить размер матрицы, скопировать старые
22    * значения и перевести позицию муравья в положительный диапазон. */
23   private void ensureFit(Position position) {
24     int shiftRow = 0;
25     int shiftColumn = 0;
26
27     /* Вычисление нового количества строк. */
28     int numRows = grid.length;
29     if (position.row < 0) {
30       shiftRow = numRows;
31       numRows *= 2;
32     } else if (position.row >= numRows) {
33       numRows *= 2;
34     }
35
36     /* Вычисление нового количества столбцов. */
37     int numColumns = grid[0].length;
38     if (position.column < 0) {
39       shiftColumn = numColumns;
40       numColumns *= 2;
41     } else if (position.column >= numColumns) {
42       numColumns *= 2;
43     }
44
45     /* При необходимости увеличить массив и изменить позицию муравья. */
46     if (numRows != grid.length || numColumns != grid[0].length) {
47       boolean[][] newGrid = new boolean[numRows][numColumns];
48       copyWithShift(grid, newGrid, shiftRow, shiftColumn);
49       ant.adjustPosition(shiftRow, shiftColumn);
50       grid = newGrid;
51     }
52   }
53
54   /* Изменение цвета ячеек. */
55   private void flip(Position position) {
56     int row = position.row;
57     int column = position.column;
58     grid[row][column] = grid[row][column] ? false : true;
59   }
```

```
60
61    /* Перемещение муравья. */
62    public void move() {
63      ant.turn(grid[ant.position.row][ant.position.column]);
64      flip(ant.position);
65      ant.move();
66      ensureFit(ant.position); // С увеличением
67    }
68
69    /* Вывод сетки. */
70    public String toString() {
71      StringBuilder sb = new StringBuilder();
72      for (int r = 0; r < grid.length; r++) {
73        for (int c = 0; c < grid[0].length; c++) {
74          if (r == ant.position.row && c == ant.position.column) {
75            sb.append(ant.orientation);
76          } else if (grid[r][c]) {
77            sb.append("X");
78       } else {
79            sb.append("_");
80       }
81      }
82        sb.append("\n");
83     }
84      sb.append("Ant: " + ant.orientation + ". \n");
85      return sb.toString();
86    }
87  }
```

Мы выделили код управления муравьем в отдельный класс. Если в системе почему-либо потребуется разместить несколько муравьев, мы сможем легко расширить код.

```
1  public class Ant {
2    public Position position = new Position(0, 0);
3    public Orientation orientation = Orientation.right;
4
5    public void turn(boolean clockwise) {
6      orientation = orientation.getTurn(clockwise);
7    }
8
9    public void move() {
10     if (orientation == Orientation.left) {
11       position.column--;
12     } else if (orientation == Orientation.right) {
13       position.column++;
14     } else if (orientation == Orientation.up) {
15       position.row--;
16     } else if (orientation == Orientation.down) {
17       position.row++;
18    }
19   }
20
21   public void adjustPosition(int shiftRow, int shiftColumn) {
22     position.row += shiftRow;
23     position.column += shiftColumn;
24   }
25 }
```

Для ориентации также определено отдельное перечисление с некоторыми вспомогательными функциями.

```
public enum Orientation {
  left, up, right, down;

  public Orientation getTurn(boolean clockwise) {
    if (this == left) {
      return clockwise ? up : down;
    } else if (this == up) {
      return clockwise ? right : left;
    } else if (this == right) {
       return clockwise ? down : up;
     } else { // down
       return clockwise ? left : right;
    }
   }

   @Override
   public String toString() {
     if (this == left) {
       return "\u2190";
     } else if (this == up) {
       return "\u2191";
     } else if (this == right) {
       return "\u2192";
     } else { // вниз
       return "\u2193";
    }
   }
}
```

Мы также определим отдельный простой класс для представления текущей позиции. С таким же успехом можно отслеживать строку и столбец по отдельности.

```
public class Position {
  public int row;
  public int column;

  public Position(int row, int column) {
    this.row = row;
    this.column = column;
  }
}
```

Такое решение работает, но оно сложнее, чем нужно.

Решение 3. HashSet

На первый взгляд «очевидно», что для представления сетки следует использовать матрицу, но на самом деле это не лучший вариант. Действительно необходим только список белых ячеек (а также позиция и ориентация муравья).

Для хранения белых ячеек можно воспользоваться коллекцией `HashSet`. Если позиция находится в `HashSet`, то текущая ячейка белая; в противном случае она черная.

При этом возникают некоторые сложности, связанные с выводом сетки. Где должен начинаться вывод? А где он должен заканчиваться?

Для вывода сетки можно хранить предполагаемые координаты левого верхнего и правого нижнего угла сетки. При каждом перемещении текущая позиция муравья сравнивается с хранимым левым верхним и правым нижним углом, которые обновляются по мере необходимости.

```
1  public class Board {
2    private HashSet<Position> whites = new HashSet<Position>();
3    private Ant ant = new Ant();
4    private Position topLeftCorner = new Position(0, 0);
5    private Position bottomRightCorner = new Position(0, 0);
6
7    public Board() { }
8
9    /* Перемещение муравья. */
10   public void move() {
11     ant.turn(isWhite(ant.position)); // Поворот
12     flip(ant.position); // Изменение цвета
13     ant.move(); // move
14     ensureFit(ant.position);
15   }
16
17   /* Изменение цвета ячеек. */
18   private void flip(Position position) {
19     if (whites.contains(position)) {
20       whites.remove(position);
21     } else {
22       whites.add(position.clone());
23    }
24   }
25
26   /* Проверка границ (левый верхний и правый нижний угол).*/
27   private void ensureFit(Position position) {
28     int row = position.row;
29     int column = position.column;
30
31     topLeftCorner.row = Math.min(topLeftCorner.row, row);
32     topLeftCorner.column = Math.min(topLeftCorner.column, column);
33
34     bottomRightCorner.row = Math.max(bottomRightCorner.row, row);
35     bottomRightCorner.column = Math.max(bottomRightCorner.column, column);
36   }
37
38   /* Проверяет, является ли ячейка белой. */
39   public boolean isWhite(Position p) {
40     return whites.contains(p);
41   }
42
43   /* Проверяет, является ли ячейка белой. */
44   public boolean isWhite(int row, int column) {
45     return whites.contains(new Position(row, column));
46   }
47
48   /* Вывод сетки. */
49   public String toString() {
```

```
50      StringBuilder sb = new StringBuilder();
51      int rowMin = topLeftCorner.row;
52      int rowMax = bottomRightCorner.row;
53      int colMin = topLeftCorner.column;
54      int colMax = bottomRightCorner.column;
55      for (int r = rowMin; r <= rowMax; r++) {
56        for (int c = colMin; c <= colMax; c++) {
57          if (r == ant.position.row && c == ant.position.column) {
58            sb.append(ant.orientation);
59          } else if (isWhite(r, c)) {
60            sb.append("X");
61       } else {
62            sb.append("_");
63       }
64      }
65        sb.append("\n");
66     }
67      sb.append("Ant: " + ant.orientation + ". \n");
68      return sb.toString();
69    }
```

Реализация `Ant` и `Orientation` остается неизменной.

Реализация `Position` слегка изменяется для поддержки функциональности `HashSet`. В качестве ключа используется позиция, поэтому необходимо реализовать функцию `hashCode()`.

```
1  public class Position {
2    public int row;
3    public int column;
4
5    public Position(int row, int column) {
6      this.row = row;
7      this.column = column;
8    }
9
10    @Override
11    public boolean equals(Object o) {
12      if (o instanceof Position) {
13        Position p = (Position) o;
14        return p.row == row && p.column == column;
15     }
16      return false;
17    }
18
19    @Override
20    public int hashCode() {
21      /* Одна из многих возможных хеш-функций. */
22      return (row * 31) ^ column;
23    }
24
25    public Position clone() {
26      return new Position(row, column);
27    }
28  }
```

Одно из преимуществ этой реализации — последовательная схема разметки строк и столбцов на тот случай, если нам потребуется обратиться к ячейке в другой точке кода.

16.23. Реализуйте метод `rand7()` на базе метода `rand5()`. Другими словами, имеется метод, генерирующий случайные числа в диапазоне от 0 до 4 (включительно). Напишите метод, который использует его для генерирования случайного числа в диапазоне от 0 до 6 (включительно).

РЕШЕНИЕ

Чтобы правильно реализовать эту функцию, необходимо позаботиться о том, чтобы каждое значение из диапазона от 0 до 6 выдавалось с вероятностью 1/7.

Первая попытка (фиксированное число вызовов)

Первый шаг — попробовать сгенерировать все числа в диапазоне от 0 до 9 и вычислить остаток от деления на 7. Код выглядит примерно так:

```
1  int rand7() {
2    int v = rand5() + rand5();
3    return v % 7;
4  }
```

К сожалению, этот код не обеспечивает равную вероятность значений. Чтобы убедиться в этом, достаточно сравнить результаты вызовов `rand5()` с результатами `rand7()`.

1-й вызов	2-й вызов	Результат		1-й вызов	2-й вызов	Результат
0	0	**0**		2	3	**5**
0	1	**1**		2	4	**6**
0	2	**2**		3	0	**3**
0	3	**3**		3	1	**4**
0	4	**4**		3	2	**5**
1	0	**1**		3	3	**6**
1	1	**2**		3	4	**0**
1	2	**3**		4	0	**4**
1	3	**4**		4	1	**5**
1	4	**5**		4	2	**6**
2	0	**2**		4	3	**0**
2	1	**3**		4	4	**1**
2	2	**4**				

Вероятность появления каждой строки — 1/25, так как `rand5()` вызывается дважды и результаты каждого вызова распределяются с вероятностью 1/5. Если подсчитать частоту появления каждого числа, можно заметить, что функция

`rand7()` будет возвращать 4 с вероятностью 5/25, а 0 — с вероятностью 3/25. Это означает, что наша функция работает неправильно; результаты не распределяются с вероятностью 1/7.

Теперь представьте, что мы изменим функцию так, чтобы использовать оператор `if`, изменим постоянный коэффициент или добавим еще один вызов `rand5()`. Мы опять получим похожую таблицу, но вероятность каждой из строк будет `1/5k`, где `k` — количество вызовов `rand5()` для конкретной строки. Разным строкам будет соответствовать разное количество вызовов.

Например, вероятность появления значения при многократном вызове функции `rand7()` будет суммой вероятностей появления числа 6 во всех строках:

$$P(\text{rand7}() = 6) = 1/5^i + 1/5^j + \ldots + 1/5^m$$

Чтобы наша функция работала правильно, эта вероятность должна составлять 1/7. Но это невозможно, поскольку 5 и 7 — взаимно простые числа, и никакая последовательность вызовов `rand5()` не может дать 1/7.

Означает ли это, что задачу решить невозможно? Не совсем. Строго говоря, это означает только то, что последовательность результатов `rand5()` не может обеспечить равномерное распределение для функции `rand7()`.

Тем не менее задачу можно решить. Нам придется использовать цикл `while` и смириться с тем, что точное количество итераций для получения результата невозможно спрогнозировать заранее.

Вторая попытка (недетерминированное количество вызовов)

Как только мы переходим к циклу `while`, работа заметно упрощается. Нужно лишь сгенерировать множество равновероятных значений (с диапазоном не менее 7 элементов). Если мы сможем сделать это, то будет достаточно отбросить элементы, превышающие наибольшее значение, кратное семи, а оставшиеся поделить с остатком на 7. Так будут получены равновероятные значения в диапазоне от 0 до 6.

В приведенном далее коде значения в диапазоне от 0 до 24 генерируются по формуле `5 * rand5() + rand5()`. Затем значения из интервала 21–24 отбрасываются, так как с ними результаты `rand7()` будет смещены в сторону от 0 до 3. Наконец, каждое значение делится на 7 (с остатком), чтобы получить равновероятные значения от 0 до 6.

Обратите внимание: поскольку мы отбрасываем значения, нет никакой гарантии, сколько вызовов `rand5()` потребуется для получения нужного значения. Именно это и понимается под *недетерминированным* количеством вызовов:

```
1  int rand7() {
2    while (true) {
3      int num = 5 * rand5() + rand5();
4      if (num < 21) {
5        return num % 7;
6      }
7    }
8  }
```

Обратите внимание на то, что формула `5 * rand5() + rand5()` предоставляет ровно один способ получения каждого числа из диапазона от 0 до 24. Это гарантирует, что все значения равновероятны.

Можно ли вместо этого использовать `2 * rand5() + rand5()`? Нет, потому что полученные значения не будут равномерно распределены: например, 6 можно получить тремя разными способами (`6 = 2*1+4`, `6 = 2*2+2`, `6 = 2*3+0`), а ноль — только одним (`0 = 2*0+0`). Таким образом, нарушится равномерное распределение значений в диапазоне.

Способ получения равномерного распределения с `2 * rand5()` *существует*, но он значительно сложнее:

```
int rand7() {
  while (true) {
    int r1 = 2 * rand5(); /* четные числа от 0 до 9 */
    int r2 = rand5(); /* используется позднее для получения 0 или 1 */
    if (r2 != 4) {
      int rand1 = r2 % 2; /* Генерируется 0 или 1 */
      int num = r1 + rand1; /* В диапазоне от 0 до 9 */
      if (num < 7) {
        return num;
      }
    }
  }
}
```

Существует бесконечное множество диапазонов, которые можно использовать. Главное, проследить за тем, чтобы диапазон был достаточно широк, а все значения равновероятны.

16.24. Разработайте алгоритм для поиска в массиве всех пар целых чисел, сумма которых равна заданному значению.

РЕШЕНИЕ

Эту задачу можно решить двумя способами. Каким — зависит от ваших предпочтений по выбору между эффективностью по времени, затратами памяти или сложностью кода.

Начнем с определения. Если мы ищем пару чисел, сумма которых равна `z`, *дополнением* `x` называется число `z-x` (то есть число, которое необходимо прибавить к `x` для получения `z`). Например, если мы ищем пару чисел, сумма которых равна 12, дополнение –5 будет равно 17.

Метод «грубой силы»

Решение методом «грубой силы» просто перебирает все пары и выводит пару в том случае, если ее сумма равна заданной целевой сумме:

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
  ArrayList<Pair> result = new ArrayList<Pair>();
  for (int i = 0 ; i < array.length; i++) {
    for (int j = i + 1; j < array.length; j++) {
      if (array[i] + array[j] == sum) {
        result.add(new Pair(array[i], array[j]));
      }
    }
  }
```

```
    return result;
}
```

Если массив содержит дубликаты (например, {5, 6, 5}), одна сумма будет выведена многократно. Обсудите этот момент с интервьюером.

Оптимизированное решение

Решение можно оптимизировать при помощи хеш-таблицы или HashSet. Работа данного алгоритма основана на переборе всего массива. Для каждого элемента x в хеш-таблице ищется sum - x и, если такое значение существует (и значение x ранее не обнаруживалось), выводится (x, sum - x). После этого алгоритм добавляет x в хеш-таблицу и переходит к следующему элементу.

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
  ArrayList<Pair> result = new ArrayList<Pair>();
  HashSet<Integer> elements = new HashSet<Integer>();
  for (int x : array) {
    int complement = sum - x;
    if (elements.contains(complement) && !elements.contains(x)) {
      result.add(new Pair(x, complement));
    }
    elements.add(x);
  }
  return result;
}
```

Это решение не выводит дубликаты, выполняется за время $O(N)$ с затратами памяти $O(N)$.

Альтернативное решение

Описанный алгоритм можно оптимизировать, если массив отсортирован. (Более того, иногда в формулировке задачи речь идет о поиске пар в отсортированном массиве.)

Допустим, имеется отсортированный массив {-2 -1 0 3 5 6 7 9 13 14}. Пусть first указывает на начало массива, а last — на его конец. Чтобы найти дополнение для first, мы перемещаем last назад, пока не найдем искомую величину. Если first + + last < sum, то дополнения к first не существует. Можно также перемещать first вперед. Тогда мы остановимся, если first окажется больше, чем last.

Почему такое решение найдет все дополнения к first? Поскольку массив отсортирован, мы проверяем меньшие числа. Если сумма first + last меньше sum, нет смысла проверять меньшие значения, они не помогут найти дополнение.

Почему эта процедура найдет все дополнения для last? Потому что все пары состоят из first и last. Мы нашли все дополнения для first, а значит, и все дополнения для last.

```
void printPairSums(int[] array, int sum) {
  Arrays.sort(array);
  int first = 0;
  int last = array.length - 1;
  while (first < last) {
    int s = array[first] + array[last];
```

```
7       if (s == sum) {
8         System.out.println(array[first] + " " + array[last]);
9         first++;
10         last--;
11       } else {
12         if (s < sum) first++;
13         else last--;
14      }
15    }
16  }
```

Алгоритм тратит время $O(N \log N)$ на сортировку и время $O(N)$ на поиск пар.

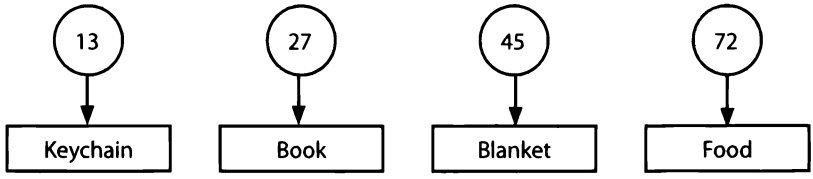
16.25. Разработайте и постройте кэш с вытеснением по давности использования (LRU, Least Recently Used). Кэш должен связывать ключи со значениями (для выполнения операций вставки и получения значения, ассоциированного с заданным ключом) и инициализироваться максимальным размером. При заполнении кэша должен вытесняться элемент, который не использовался дольше всех остальных.

РЕШЕНИЕ

Начнем с определения масштаба задачи. Какие именно функции необходимо реализовать?

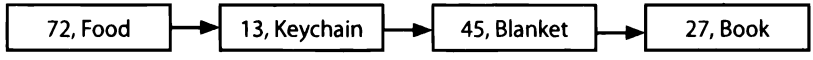
- Вставка пары «ключ/значение».
- Выборка значения по ключу.
- Нахождение самого старого элемента.
- Обновление элемента, который использовался последним.
- Вытеснение самого старого элемента при достижении емкости кэша.

Ассоциация (*ключ*, *значение*) наводит на мысль о хеш-таблице, предоставляющей простые средства поиска значения, связанного с конкретным ключом.



К сожалению, хеш-таблица обычно не предоставляет быстрых средств удаления самого старого элемента. Можно снабдить каждый элемент временно́й меткой, перебрать элементы хеш-таблицы и удалить элемент с самой старой меткой, но эта операция будет выполняться достаточно медленно ($O(N)$ для вставки).

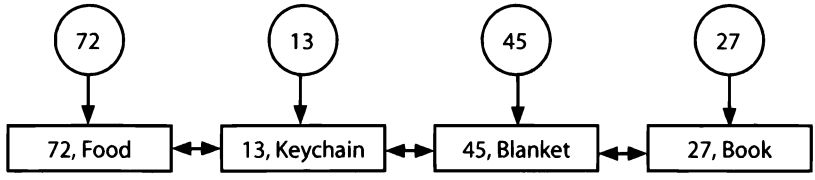
Вместо этого можно воспользоваться связным списком, упорядоченным по давности использования. Это упростит как пометку недавно использовавшихся элементов (такие элементы помещаются в начало списка), так и удаление давно использовавшихся элементов (удаление с конца списка).



К сожалению, такая реализация не предоставляет быстрых средств выборки элемента по ключу. Можно перебрать связный список и найти элемент по ключу, но это тоже будет очень медленно ($O(N)$ для выборки).

Каждый подход отлично справляется со своей половиной задачи, но ни один не справляется достаточно хорошо с единым целым. Можно ли взять все лучшее от каждого подхода? Да. Нужно использовать оба!

Связный список выглядит так же, как в предшествующем примере, но он становится двусвязным; это позволяет легко удалить элемент из середины связного списка. А хеш-таблица теперь связывает ключ с узлом связного списка, а не со значением.



Алгоритмы работают следующим образом:

- Вставка пары «ключ/значение»: создать узел связного списка с ключом и значением. Вставить узел в начало связного списка, вставить в хеш-таблицу отображение «ключ->узел».
- Выборка значения по ключу: найти узел по хеш-таблице и вернуть значение. Обновить элемент, который использовался последним (см. далее).
- Нахождение самого старого элемента: переместить узел в начало связного списка. Обновлять хеш-таблицу не обязательно.
- Вытеснение: удалить конечный элемент связного списка. Получить ключ из узла связного списка и удалить его из хеш-таблицы.

Ниже приведена реализация этих структур данных и алгоритмов.

```
1  public class Cache {
2    private int maxCacheSize;
3    private HashMap<Integer, LinkedListNode> map =
4      new HashMap<Integer, LinkedListNode>();
5    private LinkedListNode listHead = null;
6    public LinkedListNode listTail = null;
7
8    public Cache(int maxSize) {
9      maxCacheSize = maxSize;
10   }
11
12   /* Получение значения ключа и пометка недавнего использования. */
13   public String getValue(int key) {
14     LinkedListNode item = map.get(key);
15     if (item == null) return null;
16
17     /* Перемещение в начало списка. */
18     if (item != listHead) {
19       removeFromLinkedList(item);
20       insertAtFrontOfLinkedList(item);
21     }
```

```
    return item.value;
  }

  /* Удаление узла из связного списка. */
  private void removeFromLinkedList(LinkedListNode node) {
    if (node == null)  return;

    if (node.prev != null) node.prev.next = node.next;
    if (node.next != null) node.next.prev = node.prev;
    if (node == listTail) listTail = node.prev;
    if (node == listHead) listHead = node.next;
  }

  /* Вставка узла в начало связного списка. */
  private void insertAtFrontOfLinkedList(LinkedListNode node) {
    if (listHead == null) {
      listHead = node;
      listTail = node;
    } else {
      listHead.prev = node;
      node.next = listHead;
      listHead = node;
   }
  }

  /* Удаление пары ключ/значение из кэша. */
  public boolean removeKey(int key) {
    LinkedListNode node = map.get(key);
    removeFromLinkedList(node);
    map.remove(key);
    return true;
  }

  /* Занесение пары ключ/значение в кэш (с удалением
   * старого значения, если потребуется). */
  public void setKeyValue(int key, String value) {
    /* Удалить, если уже присутствует. */
    removeKey(key);

    /* Если кэш заполнен, удалить самый старый элемент. */
    if (map.size() >= maxCacheSize && listTail != null) {
      removeKey(listTail.key);
   }

    /* Insert new node. */
    LinkedListNode node = new LinkedListNode(key, value);
    insertAtFrontOfLinkedList(node);
    map.put(key, node);
  }

  private static class LinkedListNode {
    private LinkedListNode next, prev;
    public int key;
    public String value;
    public LinkedListNode(int k, String v) {
      key = k;
      value = v;
```

```
79      }
80    }
81  }
```

Класс `LinkedListNode` реализован как внутренний класс `Cache`, так как доступ к нему не нужен никаким другим классам, а сам класс существует только в границах видимости `Cache`.

16.26. Вычислите результат заданной арифметической операции, состоящей из положительных целых чисел, а также операторов +, -, * и /.

Пример:

Ввод: `2 * 3 + 5/6 * 3 + 15`

Вывод: `23.5`

РЕШЕНИЕ

Прежде всего нужно понять, что «тупое» решение — применение каждого оператора слева направо — не подойдет. Умножение и деление считаются «высокоприоритетными» операциями, которые должны выполняться перед сложением.

Например, если имеется простое выражение `3 + 6 * 2`, следует сначала выполнить умножение, а потом сложение. Если просто обрабатывать формулу слева направо, вы получите неправильный результат 18 (вместо 15).

Решение 1

Впрочем, выражение все же можно обрабатывать слева направо; просто нужно умнее подойти к процессу. Операции умножения и деления необходимо сгруппировать, чтобы они немедленно выполнялись с прилегающими операндами.

Предположим, имеется следующее выражение:

```
2 - 6 - 7*8/2 + 5
```

Ничто не помешает вычислить `2 - 6` и сохранить результат. Но когда мы встречаем `7*(`*нечто*`)`, мы знаем, что это подвыражение необходимо полностью обработать перед включением его в результат.

Для этого можно прочитать выражение слева направо с использованием двух служебных переменных.

В переменной `processing` сохраняется результат текущей группы (как оператора, так и значения). В случае сложения и вычитания группой будет текущее подвыражение. В случае умножения и деления группой будет полная последовательность (вплоть до следующего сложения или вычитания).

Если следующим подвыражением является операция сложения или вычитания (или следующего подвыражения нет), то `processing` применяется ко второй переменной `result`.

В приведенном выше примере последовательность действий будет выглядеть так:

1. Прочитать +2, применить к `processing`. Применить `processing` к `result`. Сбросить `processing`.

```
processing = {+, 2}  --> null
result = 0       --> 2
```

2. Прочитать –6, применить к processing. Применить processing к result. Сбросить processing.

```
processing = {-, 6}  --> null
result = 2       --> -4
```

3. Прочитать –7, применить к processing. Заметить, что следующим знаком является *. Продолжить.

```
processing = {-, 7}
result = -4
```

4. Прочитать *8, применить к processing. Заметить, что следующим знаком является /. Продолжить.

```
processing = {-, 56}
result = -4
```

5. Прочитать /2, применить к processing. Заметить, что следующим знаком является +, завершающий группу умножения и деления. Применить processing к result. Сбросить processing.

```
processing = {-, 28}  --> null
result = -4      --> -32
```

6. Прочитать +5, применить к processing. Применить processing к result. Сбросить processing.

```
processing = {+, 5}  --> null
result = -32     --> -27
```

Следующий код реализует этот алгоритм.

```
1  /* Вычисление результата серии арифметических операций. Входные данные
2   * читаются слева направо и применяются к result. Если в потоке встречается
3   * умножение или деление, операция применяется к временной переменной. */
4  double compute(String sequence) {
5    ArrayList<Term> terms = Term.parseTermSequence(sequence);
6    if (terms == null) return Integer.MIN_VALUE;
7
8    double result = 0;
9    Term processing = null;
10   for (int i = 0; i < terms.size(); i++) {
11     Term current = terms.get(i);
12     Term next = i + 1 < terms.size() ? terms.get(i + 1) : null;
13
14     /* Применение текущей операции к processing. */
15     processing = collapseTerm(processing, current);
16
17     /* Если следующей операцией является + или -, обработка группы
18      * завершается, и processing применяется к result. */
19     if (next == null || next.getOperator() == Operator.ADD
20         || next.getOperator() == Operator.SUBTRACT) {
21       result = applyOp(result, processing.getOperator(), processing.getNumber());
22       processing = null;
23     }
24   }
```

```
25
26    return result;
27  }
28
29  /* Объединение двух подвыражений с использованием оператора secondary
30   * и операндов из обоих подвыражений. */
31  Term collapseTerm(Term primary, Term secondary) {
32    if (primary == null) return secondary;
33    if (secondary == null) return primary;
34
35    double value = applyOp(primary.getNumber(), secondary.getOperator(),
36                           secondary.getNumber());
37    primary.setNumber(value);
38    return primary;
39  }
40
41  double applyOp(double left, Operator op, double right) {
42    if (op == Operator.ADD) return left + right;
43    else if (op == Operator.SUBTRACT) return left - right;
44    else if (op == Operator.MULTIPLY) return left * right;
45    else if (op == Operator.DIVIDE) return left / right;
46    else return right;
47  }
48
49  public class Term {
50    public enum Operator {
51      ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
52    }
53
54    private double value;
55    private Operator operator = Operator.BLANK;
56
57    public Term(double v, Operator op) {
58      value = v;
59      operator = op;
60    }
61
62    public double getNumber() { return value; }
63    public Operator getOperator() { return operator; }
64    public void setNumber(double v) { value = v; }
65
66    /* Серия арифметических операций разбирается в список Term.
67     * Например, 3-5*6 : [{BLANK,3}, {SUBTRACT, 5}, {MULTIPLY, 6}].
68     * При ошибке форматирования возвращается null. */
69    public static ArrayList<Term> parseTermSequence(String sequence) {
70      /* Код находится в архиве примеров. */
71    }
72  }
```

Решение выполняется за время $O(N)$, где N — длина исходной строки.

Решение 2

Альтернативное решение этой задачи основано на использовании двух стеков: для чисел и для операторов.

Обработка происходит следующим образом:

- Каждый раз, когда мы встречаем число, оно заносится в стек чисел `numberStack`.
- Операторы заносятся в стек `operatorStack` — при условии, что оператор имеет более высокий приоритет, чем текущая вершина стека. Если `priority(currentOperator) <= priority(operatorStack.top())`, происходит «свертка» вершин стеков:
  - *Свертка*: из `numberStack` извлекаются два элемента, из `operatorStack` извлекается оператор; оператор применяется к числам, а результат заносится в `numberStack`.
  - *Приоритет*: операции сложения и вычитания имеют одинаковый приоритет, который ниже приоритета умножения и деления (эти две операции также имеют равный приоритет).

Свертка продолжается до тех пор, пока приведенное выше неравенство не будет нарушено. В этот момент `currentOperator` заносится в `operatorStack`.

- В самом конце выполняется свертка стека.

Рассмотрим работу алгоритма на примере: `2 - 6 - 7 * 8 / 2 + 5`

	Действие	numberStack	operatorStack
2	numberStack.push(2)	2	[пусто]
–	operatorStack.push(-)	2	–
6	numberStack.push(6)	6, 2	–
–	collapseStacks [2 - 6] operatorStack.push(-)	–4 –4	[пусто]
7	numberStack.push(7)	7, –4	–
*	operatorStack.push(*)	7, -4	*, –
8	numberStack.push(8)	8, 7, -4	*, –
/	collapseStack [7 * 8] operator Stack.push(/)	56, –4 56, –4	– /, –
2	numberStack.push(2)	2, 56, –4	/, –
+	collapseStack [56 / 2] collapseStack [-4 - 28] operatorStack.push(+)	28, -4 -32 -32	- [пусто] +
5	numberStack.push(5)	5, -32	+
	collapseStack [-32 + 5]	–27	[пусто]
	return –27		

Следующий код реализует этот алгоритм.

```
1  public enum Operator {
2    ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
3  }
```

```
4
5  double compute(String sequence) {
6    Stack<Double> numberStack = new Stack<Double>();
7    Stack<Operator> operatorStack = new Stack<Operator>();
8
9    for (int i = 0; i < sequence.length(); i++) {
10     try {
11       /* Получить число и занести в стек. */
12       int value = parseNextNumber(sequence, i);
13       numberStack.push((double) value);
14
15       /* Перейти к оператору. */
16       i += Integer.toString(value).length();
17       if (i >= sequence.length()) {
18         break;
19     }
20
21       /* Получить и занести оператор в стек (со сверткой, если потребуется) */
22       Operator op = parseNextOperator(sequence, i);
23       collapseTop(op, numberStack, operatorStack);
24       operatorStack.push(op);
25     } catch (NumberFormatException ex) {
26       return Integer.MIN_VALUE;
27    }
28   }
29
30   /* Завершающая свертка. */
31   collapseTop(Operator.BLANK, numberStack, operatorStack);
32   if (numberStack.size() == 1 && operatorStack.size() == 0) {
33     return numberStack.pop();
34   }
35   return 0;
36 }
37
38 /* Выполнять свертку до момента priority(futureTop) > priority(top).
39  * При свертке извлекаются 2 верхних числа, к ним применяется оператор
40  * со стека операторов, а результат заносится обратно в стек чисел.*/
41 void collapseTop(Operator futureTop, Stack<Double> numberStack,
42                  Stack<Operator> operatorStack) {
43   while (operatorStack.size() >= 1 && numberStack.size() >= 2) {
44     if (priorityOfOperator(futureTop) <=
45         priorityOfOperator(operatorStack.peek())) {
46       double second = numberStack.pop();
47       double first = numberStack.pop();
48       Operator op = operatorStack.pop();
49       double collapsed = applyOp(first, op, second);
50       numberStack.push(collapsed);
51     } else {
52       break;
53    }
54   }
55 }
56
57 /* Получение приоритета оператора:
58  *     сложение == вычитание < умножение == деление. */
59 int priorityOfOperator(Operator op) {
60   switch (op) {
```

```
61      case ADD: return 1;
62      case SUBTRACT: return 1;
63      case MULTIPLY: return 2;
64      case DIVIDE: return 2;
65      case BLANK: return 0;
66    }
67    return 0;
68  }
69
70  /* Применение оператора: left [op] right. */
71  double applyOp(double left, Operator op, double right) {
72    if (op == Operator.ADD) return left + right;
73    else if (op == Operator.SUBTRACT) return left - right;
74    else if (op == Operator.MULTIPLY) return left * right;
75    else if (op == Operator.DIVIDE) return left / right;
76    else return right;
77  }
78
79  /* Возвращает число, начинающееся со смещения offset. */
80  int parseNextNumber(String seq, int offset) {
81    StringBuilder sb = new StringBuilder();
82    while (offset < seq.length() && Character.isDigit(seq.charAt(offset))) {
83      sb.append(seq.charAt(offset));
84      offset++;
85    }
86    return Integer.parseInt(sb.toString());
87  }
88
89  /* Возвращает оператор, начинающийся со смещения offset. */
90  Operator parseNextOperator(String sequence, int offset) {
91    if (offset < sequence.length()) {
92      char op = sequence.charAt(offset);
93      switch(op) {
94        case '+': return Operator.ADD;
95        case '-': return Operator.SUBTRACT;
96        case '*': return Operator.MULTIPLY;
97        case '/': return Operator.DIVIDE;
98      }
99    }
100    return Operator.BLANK;
101  }
```

Этот код выполняется за время $O(N)$, где N — длина строки.

В этом решении задействован большой объем рутинного кода разбора строк. Помните, что все эти подробности на собеседовании не так уж важны. Собственно, интервьюер даже может указать, что выражение заранее разобрано в некую структуру данных.

Сосредоточьтесь на модуляризации кода и «вынесении» рутинных и менее интересных частей кода в другие функции. Главное — добиться того, чтобы заработала основная функция `compute`, а подробности подождут!

17

Сложные задачи

17.1. Напишите функцию суммирования двух чисел без использования «+» или любых других арифметических операторов.

РЕШЕНИЕ

Первое, что приходит в голову в подобных задачах, — поразрядные операции. Почему? Если нельзя использовать оператор «+», то что еще остается? Будем суммировать числа так, как это делают компьютеры!

Теперь нужно поглубже разобраться в том, как работает суммирование. Разберем дополнительный пример и попробуем найти нечто интересное — выявить закономерность, которую можно было бы воспроизвести в коде.

Итак, рассмотрим дополнительную задачу. Для удобства будем использовать десятичную систему счисления.

Чтобы просуммировать `759 + 674`, мы обычно складываем `digit[0]` из каждого числа, переносим единицу, затем переходим к `digit[1]`, переносим и т. д. Точно так же можно работать с битами: просуммировать все разряды и при необходимости сделать переносы единиц.

Можно ли упростить алгоритм? Да! Допустим, я хочу разделить «суммирование» и «перенос». Мне придется проделать следующее.

1. Выполнить операцию `759 + 674`, «забыв» о переносе. В результате получится `323`.
2. Выполнить операцию `759 + 674`, но сделать только переносы (без суммирования разрядов). В результате получится `1110`.
3. Теперь нужно сложить результаты первых двух операций (используя тот же механизм, описанный в шагах 1 и 2): `1110 + 323 = 1433`.

Теперь вернемся к двоичной системе.

1. Если просуммировать пару двоичных чисел, без учета переноса знака, то `i`-й бит суммы может быть нулевым, только если `i`-е биты чисел `a` и `b` совпадали (оба имели значение 0 или 1). Это классическая операция `XOR`.
2. Если суммировать пару чисел, выполняя *только* перенос, то `i`-й бит суммы будет равен 1, только если `i-1`-е биты обоих чисел (`a` и `b`) имели значение 1. Это операция `AND` со сдвигом.
3. Повторять, пока остаются переносы.

Следующий код реализует данный алгоритм.

```
1  int add(int a, int b) {
2    if (b == 0) return a;
```

```
3    int sum = a ^ b; // суммирование без переноса
4    int carry = (a & b) << 1; // перенос без суммирования
5    return add(sum, carry); // повторить с sum + carry
6  }
```

Также возможна итеративная реализация.

```
1  int add(int a, int b) {
2    while (b != 0) {
3      int sum = a ^ b; // суммирование без переноса
4      int carry = (a & b) << 1; // перенос без суммирования
5      a = sum;
6      b = carry;
7    }
8    return a;
9  }
```

Задачи, связанные с реализацией базовых операций (сложение, вычитание), встречаются относительно часто. Чтобы решить такую задачу, нужно досконально разобраться с тем, как обычно выполняется такая операция, а потом реализовать ее заново с учетом поставленных ограничений.

17.2. Напишите метод, моделирующий тасование карточной колоды. Колода должна быть идеально перетасована — иначе говоря, все 52! перестановки карт должны быть равновероятными. Предполагается, что у вас в распоряжении имеется идеальный генератор случайных чисел.

РЕШЕНИЕ

Это задача очень популярна на собеседованиях, а алгоритм для ее решения широко известен. Если вы еще не принадлежите к числу счастливчиков, знакомых с алгоритмом, читайте дальше.

Возьмем массив из `n` элементов. Предположим, он выглядит так:

```
[1] [2] [3] [4] [5]
```

Воспользуемся методом «базовый случай с расширением»: допустим, имеется метод `shuffle(...)`, который работает с `n - 1` элементами. Можно ли использовать его для перетасовки `n` элементов?

Конечно. Более того, это несложно. Нужно сначала перетасовать первые `n - 1` элементов, а потом взять `n`-й элемент и поменять его местами со случайным элементом массива. И все!

В рекурсивной реализации этот алгоритм выглядит так:

```
1  /* Случайное число в диапазоне от lower до higher включительно */
2  int rand(int lower, int higher) {
3    return lower + (int)(Math.random() * (higher - lower + 1));
4  }
5
6  int[] shuffleArrayRecursively(int[] cards, int i) {
7    if (i == 0) return cards;
8
9    shuffleArrayRecursively(cards, i - 1); // Перетасовать предыдущую часть
10    int k = rand(0, i); // Выбрать случайный индекс
```

```
11
12    /* Переставить местами элементы k и i */
13    int temp = cards[k];
14    cards[k] = cards[i];
15    cards[i] = temp;
16
17    /* Вернуть перетасованный массив */
18    return cards;
19  }
```

А как этот алгоритм будет выглядеть в итеративной версии? Поразмыслите над этим. В сущности, алгоритм просто перемещается по массиву, и для каждого элемента `i` меняет местами `array[i]` со случайным элементом от 0 до `i` включительно.

Этот алгоритм очень четко реализуется в итеративной форме:

```
1  void shuffleArrayIteratively(int[] cards) {
2    for (int i = 0; i < cards.length; i++) {
3      int k = rand(0, i);
4      int temp = cards[k];
5      cards[k] = cards[i];
6      cards[i] = temp;
7    }
8  }
```

Чаще всего на собеседованиях предлагается именно такое итеративное решение.

17.3. Напишите метод, генерирующий случайное множество из `m` целых чисел из массива размером `n`. Все элементы должны выбираться с одинаковой вероятностью.

РЕШЕНИЕ

Как и в предыдущей похожей задаче (17.2), можно воспользоваться рекурсивным подходом с использованием метода «базовый случай и расширение».

Допустим, имеется алгоритм, который умеет извлекать случайное множество из `m` элементов из массива размером `n-1`. Как использовать этот алгоритм для выборки случайного множества из `m` элементов из массива размером `n`?

Сначала мы извлечем случайное множество размером `m` из первых `n-1` элементов. Затем нужно решить, должен ли элемент `array[n]` вставляться в подмножество (для этого потребуется извлечь из него случайный элемент).

Для этого проще всего выбрать случайное число `k` от 0 до `n`. Если `k < m`, то `array[n]` вставляется в `subset[k]`. Таким образом элемент `array[n]` будет «справедливо» (то есть с пропорциональной вероятностью) вставлен в подмножество, а из подмножества будет «справедливо» удален случайный элемент.

Псевдокод этого рекурсивного алгоритма выглядит так:

```
1  int[] pickMRecursively(int[] original, int m, int i) {
2    if (i + 1 == m) { // Базовый случай
3      /* Вернуть первые m элементов original */
4    } else if (i + 1 > m) {
5      int[] subset = pickMRecursively(original, m, i - 1);
6      int k =  random value between 0 and i, inclusive
```

```
7      if (k < m) {
8        subset[k] = original[i];
9      }
10       return subset;
11     }
12     return null;
13   }
```

В итеративной форме алгоритм записывается еще проще. В этом случае массив `subset` инициализируется первыми `m` элементами `original`. Далее мы перебираем массив, начиная с элемента `m`, и вставляем `array[i]` в подмножество в (случайной) позиции `k` для всех `k < m`.

```
1  int[] pickMIteratively(int[] original, int m) {
2    int[] subset = new int[m];
3
4    /* Заполнение массива subset первой частью исходного массива */
5    for (int i = 0; i < m ; i++) {
6      subset[i] = original[i];
7    }
8
9    /* Перебор остальных элементов массива original. */
10    for (int i = m; i < original.length; i++) {
11      int k = rand(0, i); // Случайное число от 0 до i включительно
12      if (k < m) {
13        subset[k] = original[i];
14      }
15    }
16
17    return subset;
18  }
```

Как и следовало ожидать, оба решения очень похожи на алгоритм тасования массива.

17.4. Массив A содержит все целые числа от 0 до n, кроме одного. Считается, что обратиться ко всему целому числу в A за одну операцию невозможно. Элементы A представлены в двоичной форме, и для работы с ними может использоваться только одна операция — «получить j-й бит элемента A[i]», — выполняемая за постоянное время. Напишите код для поиска отсутствующего целого числа. Удастся ли вам выполнить поиск за время $O(n)$?

РЕШЕНИЕ

Возможно, вам уже попадалась очень похожая задача: дан список чисел от `0` до `n`, одно число отсутствует, нужно его найти. Для этого достаточно подсчитать сумму чисел в имеющемся ряду и сравнить ее с суммой чисел от `0` до `n`, которая равна `n * (n + 1)/2`. Разница и будет отсутствующим числом.

Задачу можно решить вычислением значения каждого числа по его двоичному представлению и оопределением суммы.

Время выполнения этого решения пропорционально `n * length(n)`, где `length` — количество битов в `n`. Обратите внимание, что `length(n) = log`$_2$`(n)`. Итак, фактическое время выполнения составляет `O(n log(n))`, что не впечатляет.

Как еще можно решить эту задачу?

Можно использовать аналогичный подход, но работать напрямую с битами.

Рассмотрим список двоичных чисел (---- соответствует отсутствующему значению).

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Отсутствие числа создает дисбаланс единиц и нулей в младшем разряде, этот разряд мы обозначим LSB_1. В списке чисел от 0 до n должно присутствовать одинаковое количество нулей и единиц (при нечетном n) или дополнительный 0, если n — четное. То есть:

- если `n % 2 == 1`, то `count(0s) = count(1s)`;
- если `n % 2 == 0`, то `count(0s) = 1 + count(1s)`.

Это означает, что `count(0s)` всегда будет больше или равно `count(1s)`.

Когда мы удаляем значение v из списка, чтобы узнать, является ли v четным или нечетным, достаточно проверить младший бит остальных чисел в списке.

	`n % 2 == 0` `count(0s) = 1 + count(1s)`	`n % 2 == 1` `count(0s) = count(1s)`
`v % 2 == 0` LSB_1`(v) = 0`	0 удален. count(0s) = count(1s)	0 удален. count(0s) < count(1s)
`v % 2 == 1` LSB_1`(v) = 1`	1 удалена. count(0s) > count(1s)	1 удалена. count(0s) > count(1s)

Итак, если `count(0s) <= count(1s)`, то v — четное. Если `count(0s) > count(1s)`, то v — нечетное. Теперь мы можем исключить все четные значения и сосредоточиться на нечетных или удалить все нечетные и сосредоточиться на четных.

Но как узнать следующий бит v? Если v находилось бы в (сокращенном) списке, то следующий бит был бы равен:

$count_2(0s) = count_2(1s)$ OR $count_2(0s) = 1 + count_2(1s)$,

где $count_2$ – количество 0 или 1 в LSB_2.

Как в предыдущем примере, можно узнать значение второго младшего бита v:

	$count_2(0s) = 1 + count_2(1s)$	$count_2(0s) = count_2(1s)$
$LSB_2(v) == 0$	0 удален. $count_2(0s) = count_2(1s)$	0 удален. $count_2(0s) < count_2(1s)$
$LSB_2(v) == 1$	1 удалена. $count_2(0s) > count_2(1s)$	1 удалена. $count_2(0s) > count_2(1s)$

Мы снова приходим к тем же выводам:

- если $count_2(0s) <= count_2(1s)$, то $LSB_2(v) = 0$;
- если $count_2(0s) > count_2(1s)$, то $LSB_2(v) = 1$.

Этот процесс можно повторить для каждого бита. На каждом шаге мы подсчитываем количество нулей и единиц в i, чтобы узнать, является $LSB_i(v)$ нулем или единицей. Затем мы отбрасываем значения, для которых $LSB_i(x) != LSB_i(v)$. Если v четное, мы отбрасываем нечетные числа и т. д.

По окончании этого процесса все биты в v будут вычислены. На каждой успешной итерации мы проверяем n, затем n/2, затем n/4 и т. д. При таком подходе время выполнения составит $O(N)$.

Чтобы происходящее стало более понятным, возьмем конкретный пример. На первой итерации мы начинаем с чисел:

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Поскольку $count_1(0s) > count_1(1s)$, то $LSB_1(v) = 1$. Отбросим все x, где $LSB_1(x) != LSB_1(v)$.

~~00000~~	~~00100~~	~~01000~~	~~01100~~
00001	00101	01001	01101
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

Теперь $count_2(0s) > count_2(1s)$, и мы знаем, что $LSB_2(v) = 1$. Отбросим все x, где $LSB_2(x) != LSB_2(v)$.

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	01101
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

На этот раз $count_3(0s) <= count_3(1s)$. Мы знаем, что $LSB_3(v) = 0$. Отбросим все x, где $LSB_3(x) != LSB_3(v)$.

~~00000~~	~~00100~~	~~01000~~	~~01100~~
~~00001~~	~~00101~~	~~01001~~	01101
~~00010~~	~~00110~~	~~01010~~	
-----	00111	01011	

Осталось единственное число. В этом случае $count_4(0s) <= count_4(1s)$, поэтому $LSB_4(v) = 0$.

Когда все числа, где $LSB_4(v) != 0$, отброшены, список оказывается пустым. Как только список пуст, $count_i(0s) <= count_i(1s)$. Итак, $LSB_i(v) = 0$. Получив пустой список, мы можем заполнить оставшиеся биты v нулями.

Этот процесс позволяет найти значение v. Для нашего случая получаем v = 00011.

Ниже приведена реализация описанного алгоритма.

```
1  int findMissing(ArrayList<BitInteger> array) {
2    /* Начать с младшего бита и продвигаться вверх */
3    return findMissing(array, 0);
```

```
4  }
5
6  int findMissing(ArrayList<BitInteger> input, int column) {
7    if (column >= BitInteger.INTEGER_SIZE) { // Готово!
8      return 0;
9    }
10    ArrayList<BitInteger> oneBits = new ArrayList<BitInteger>(input.size()/2);
11    ArrayList<BitInteger> zeroBits = new ArrayList<BitInteger>(input.size()/2);
12
13    for (BitInteger t : input) {
14      if (t.fetch(column) == 0) {
15        zeroBits.add(t);
16      } else {
17        oneBits.add(t);
18     }
19    }
20    if (zeroBits.size() <= oneBits.size()) {
21      int v = findMissing(zeroBits, column + 1);
22      return (v << 1) | 0;
23    } else {
24      int v = findMissing(oneBits, column + 1);
25      return (v << 1) | 1;
26    }
27  }
```

В строках с 24 по 27 рекурсивно рассчитываются другие биты v. Затем вставляется бит 0 или 1 в зависимости от того, истинно ли условие $\texttt{count}_1\texttt{(0s) <= count}_1\texttt{(1s)}$.

17.5. Имеется массив, заполненный буквами и цифрами. Найдите самый длинный подмассив с равным количеством букв и цифр.

РЕШЕНИЕ

Во введении обсуждалась важность создания хороших, универсальных примеров — и это абсолютно верно. Однако не менее важно понимать, что действительно важно в формулировке конкретной задачи.

В данном случае требуется только одно: равное количество букв и цифр. Все буквы рассматриваются одинаково, все цифры рассматриваются одинаково. Следовательно, в примере можно ограничиться одной буквой и одной цифрой, скажем, *A* или *B*, 0 или 1 и т. д.

С учетом этого факта рассмотрим пример:

```
[A, B, A, A, A, B, B, B, A, B, A, A, B, B, A, A, A, A, A, A]
```

Мы ищем наибольший подмассив `subarray`, для которого `count(A, subarray)` `count(B, subarray)`.

Метод «грубой силы»

Начнем с самого тривиального решения: просто перебрать все возможные подмассивы, подсчитать количество *A* и *B* (или букв или цифр) и найти самую длинную последовательность с равным количеством вхождений.

В эту процедуру можно внести одну небольшую оптимизацию: можно начать с самого длинного подмассива, и как только будет найден подмассив, для которого выполняется условие равенства, — вернуть его.

```
1  /* Вернуть самый длинный подмассив с равными количествами 0 и 1.
2   * Проверять подмассивы начиная с самого длинного. Как только такой
3   * подмассив будет найден, вернуть его. */
4  char[] findLongestSubarray(char[] array) {
5    for (int len = array.length; len > 1; len--) {
6      for (int i = 0; i <= array.length - len; i++) {
7        if (hasEqualLettersNumbers(array, i, i + len - 1)) {
8          return extractSubarray(array, i, i + len - 1);
9      }
10     }
11   }
12   return null;
13 }
14
15 /* Проверить, содержит ли подмассив равное количество цифр и букв. */
16 boolean hasEqualLettersNumbers(char[] array, int start, int end) {
17   int counter = 0;
18   for (int i = start; i <= end; i++) {
19     if (Character.isLetter(array[i])) {
20       counter++;
21     } else if (Character.isDigit(array[i])) {
22       counter--;
23     }
24   }
25   return counter == 0;
26 }
27
28 /* Вернуть подмассив из элементов от start до end (включительно). */
29 char[] extractSubarray(char[] array, int start, int end) {
30   char[] subarray = new char[end - start + 1];
31   for (int i = start; i <= end; i++) {
32     subarray[i - start] = array[i];
33   }
34   return subarray;
35 }
```

Несмотря на оптимизацию, этот алгоритм по-прежнему выполняется за время $O(N^2)$, где N — длина массива.

Оптимальное решение

Сейчас мы ищем подмассив, в котором количество букв равно количеству цифр. А если начать с начала, подсчитывая буквы и цифры?

	a	a	a	a	1	1	a	1	1	a	a	1	a	a	1	a	a	a	a	a
#a	1	2	3	4	4	4	5	5	5	6	7	7	8	9	9	10	11	12	13	14
#1	0	0	0	0	1	2	2	3	4	4	4	5	5	5	6	6	6	6	6	6

Конечно, для любой позиции, в которой количество букв равно количеству цифр, подмассив от индекса 0 до этого индекса является «равным».

Однако так будут найдены только «равные» подмассивы, начинающиеся с индекса 0. Как найти другие подмассивы?

Представьте ситуацию: допустим, «равный» подмассив (`a11a1a`) вставляется после массива `a1aaa1`. Как это повлияет на счетчики?

	a	1	a	a	a	1	|	a	1	1	a	1	a
#a	1	1	2	3	4	4	|	5	5	5	6	6	7
#1	0	1	1	1	1	2	|	2	3	4	4	5	5

Проанализируем значения до конца подмассива (4, 2) и конечной позиции (7, 5). Хотя числа отличаются, разности остаются неизменными: 4 – 2 = 7 – 5. И это вполне понятно, так как добавляемое количество букв и цифр одинаково, разность не должна изменяться.

Заметим, что когда разности одинаковы, подмассив начинается в позиции после начального индекса совпадения и продолжается до завершающего индекса совпадения. Это объясняет строку 10 в следующем коде.

Дополним приведенный выше массив разностями.

	a	a	a	a	1	1	a	1	1	a	a	1	a	a	1	a	a	a	a	a
#a	1	2	3	4	4	4	5	5	5	6	7	7	8	9	9	10	11	12	13	14
#1	0	0	0	0	1	2	2	3	4	4	4	5	5	5	6	6	6	6	6	6
–	1	2	3	4	3	2	3	2	1	2	3	2	3	4	3	4	5	6	7	8

Если возвращаемые разности совпадают, мы знаем, что обнаружен «равный» подмассив. Чтобы найти самый длинный подмасив, следует искать два одинаковых значения, удаленные на максимальное расстояние.

Для этого первое вхождение каждой конкретной разности будет сохраняться в хеш-таблице. Затем при каждом последующем обнаружении той же разности мы смотрим, превосходит ли текущий подмассив (от первого вхождения до текущего индекса) ранее обнаруженный максимум. Если текущий подмассив оказывается длиннее, максимум обновляется.

```
1  char[] findLongestSubarray(char[] array) {
2    /* Вычисление разностей между количествами букв и цифр. */
3    int[] deltas = computeDeltaArray(array);
4
5    /* Поиск пар с равными значениями на наибольшем расстоянии. */
6    int[] match = findLongestMatch(deltas);
7
8    /* Возвращение подмассива. Обратите внимание: подмассив начинается
9     *  со СЛЕДУЮЩЕЙ позиции за первым вхождением разности. */
10   return extract(array, match[0] + 1, match[1]);
11 }
12
13 /* Вычисление разности между количеством букв и цифр
14  * от начала массива до каждого индекса. */
15 int[] computeDeltaArray(char[] array) {
16   int[] deltas = new int[array.length];
17   int delta = 0;
18   for (int i = 0; i < array.length; i++) {
```

```
19      if (Character.isLetter(array[i])) {
20        delta++;
21      } else if (Character.isDigit(array[i])) {
22        delta--;
23     }
24      deltas[i] = delta;
25    }
26    return deltas;
27  }
28
29  /* Поиск в массиве разностей пары одинаковых значений
30   * с наибольшей разностью индексов. */
31  int[] findLongestMatch(int[] deltas) {
32    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
33    map.put(0,  -1);
34    int[] max = new int[2];
35    for (int i = 0; i < deltas.length; i++) {
36      if (!map.containsKey(deltas[i])) {
37        map.put(deltas[i],  i);
38      } else {
39        int match = map.get(deltas[i]);
40        int distance = i - match;
41        int longest = max[1] - max[0];
42        if (distance > longest) {
43          max[1] = i;
44          max[0] = match;
45      }
46     }
47    }
48    return max;
49  }
50
51  char[] extract(char[] array, int start, int end) { /* То же */ }
```

Решение выполняется за время $O(N)$, где N — размер массива.

17.6. Напишите метод, который будет подсчитывать количество цифр «2» в записи чисел от 0 до n (включительно).

Пример:

Ввод: 25

Вывод: 9 (2, 12, 20, 21, 22, 23, 24 и 25. Обратите внимание: в числе 22 — две двойки).

РЕШЕНИЕ

Начать можно с решения методом «грубой силы». Возможно, с него даже *нужно* начать. Помните, что интервьюеры хотят видеть, как вы подходите к решению задачи. «Лобовое» решение станет хорошей отправной точкой.

```
1  /* Подсчет количества цифр '2' от 0 до n */
2  int numberOf2sInRange(int n) {
3    int count = 0;
4    for (int i = 2; i <= n; i++) { // Начать можно с 2
5      count += numberOf2s(i);
```

```
6    }
7    return count;
8  }
9
10 /* Подсчет цифр '2' в одном числе */
11 int number0f2s(int n) {
12   int count = 0;
13   while (n > 0) {
14     if (n % 10 == 2) {
15       count++;
16     }
17     n = n / 10;
18   }
19   return count;
20 }
```

Если здесь что-то и заслуживает внимания, так это выделение `number0f2s` в отдельный метод. Этим вы продемонстрируете внимание к логической четкости кода.

Усовершенствованное решение

Можно рассматривать задачу не с точки зрения диапазонов чисел, а с точки зрения разрядов — цифра за цифрой.

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
...
110 111 112 113 114 115 116 117 118 119
```

Мы знаем, что в последовательном ряду из десяти чисел последний разряд принимает значение 2 только один раз. И вообще, любой разряд может быть равен 2 приблизительно один раз из десяти.

Мы говорим «приблизительно», потому что необходимо учитывать граничные условия. Например, от 1 до 100 цифра десятков равна 2 ровно в 10% случаев. С другой стороны, в диапазоне 1–37 в разряде десятков цифра 2 встречается намного чаще.

Точное количество двоек можно вычислить, рассмотрев все по отдельности три случая: `digit < 2`, `digit = 2` и `digit > 2`.

Случай: `digit < 2`

Если `x = 61523` и `d = 3`, то `x[d] = 1` (это означает, что `d`-й разряд `x` равен `1`). Рассмотрим двойки, находящиеся в 3-м разряде, в диапазонах `2000-2999`, `12000-12999`, `22000-22999`, `32000-32999`, `42000-42999` и `52000-52999`. Диапазон `62000-62999` еще не достигнут, поэтому в 3-м разряде всего насчитывается 6000 двоек. Такое же количество можно получить, если просто подсчитать все двойки в 3-м разряде в диапазоне чисел от 1 до 6000.

Другими словами, чтобы рассчитать количество двоек в `d`-м разряде, достаточно округлить значение *в меньшую сторону* до 10^{d+1}, а затем разделить на 10.

```
если x[d] < 2: count2sInRangeAtDigit(x, d) =
    y = округлить вниз до ближайшего 10^(d+1)
вернуть y / 10
```

Случай: digit > 2

Давайте рассмотрим случай, когда значение d-го разряда больше 2 (x[d]>2). Если использовать ту же логику, становится понятно, что количество двоек в 3-м разряде диапазона 0-63525 будет таким же, как в диапазоне 0-7000. Таким образом, вместо округления вниз мы будем округлять вверх.

```
если x[d] > 2: count2sInRangeAtDigit(x, d) =
    y = округлить вверх до ближайшего 10^(d+1)
return y / 10
```

Случай: digit = 2

Последний случай самый трудный, но мы можем использовать ту же логику. Пусть x = 62523 и d = 3. Мы знаем, что диапазоны не изменились (2000-2999, 12000-12999, ..., 52000-52999). Сколько двоек может появиться в 3-м разряде в диапазоне 62000-62523? Подсчитать несложно — 524 (62000, 62001, ..., 62523).

```
если x[d] > 2: count2sInRangeAtDigit(x, d) =
    y = округлить вниз до 10d+1
    z = правая сторона x (т.е. x % 10d)
    вернуть y / 10 + z + 1
```

Теперь нужно перебрать все цифры в числе. Реализация данного кода относительно проста:

```
1  int count2sInRangeAtDigit(int number, int d) {
2    int powerOf10 = (int) Math.pow(10, d);
3    int nextPowerOf10 = powerOf10 * 10;
4    int right = number % powerOf10;
5
6    int roundDown = number - number % nextPowerOf10;
7    int roundUp = roundDown + nextPowerOf10;
8
9    int digit = (number / powerOf10) % 10;
10    if (digit < 2) { // Один из трех случаев
11      return roundDown / 10;
12    } else if (digit == 2) {
13      return roundDown / 10 + right + 1;
14    } else {
15      return roundUp / 10;
16    }
17  }
18
19  int count2sInRange(int number) {
20    int count = 0;
21    int len = String.valueOf(number).length();
22    for (int digit = 0; digit < len; digit++) {
23      count += count2sInRangeAtDigit(number, digit);
24    }
25    return count;
26  }
```

Данная задача требует тщательного тестирования. Убедитесь, что вы знаете все граничные случаи и проверили каждый из них.

17.7. Правительство ежегодно публикует список 10 000 самых распространенных имен, выбираемых родителями, и их частоты (количество детей с каждым именем). Единственная проблема заключается в том, что некоторые имена существуют в нескольких вариантах написания. Например, «John» и «Jon» — фактически одно имя, но в списке они будут занимать две разные позиции. Для двух заданных списков (один содержит имена/частоты, другой — пары эквивалентных имен) напишите алгоритм для вывода нового списка истинной частоты каждого имени. Обратите внимание: если John и Jon являются синонимами, и при этом Jon и Johnny являются синонимами, то и John и Johnny являются синонимами (отношение транзитивно и симметрично). В окончательном списке в качестве «настоящего» может использоваться любое имя.

Пример:

Ввод:

Имена: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Синонимы: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Вывод: John (27), Kris (36)

РЕШЕНИЕ

Начнем с хорошего примера: одни имена должны иметь несколько синонимов, а другие — ни одного. Кроме того, список синонимов должен быть достаточно разнообразным в отношении того, какие имена находятся в левой части, а какие в правой; например, имя Johnny не должно встречаться только в левой части, так как мы создаем группу (John, Jonathan, Jon, Johnny).

Следующий список подойдет для наших целей.

Имя	Количество
John	10
Jon	3
Davis	2
Kari	3
Johnny	11
Carlton	8
Carleton	2
Jonathan	9
Carrie	5

Имя	Синоним
Jonathan	John
Jon	Johnny
Johnny	John
Kari	Carrie
Carleton	Carlton

Итоговый список должен выглядеть примерно так: John (33), Kari (8), Davis (2), Carleton (10).

Решение 1

Допустим, список имен передается в виде хеш-таблицы (а если нет, ее можно достаточно легко построить).

Начинаем читать пары из списка синонимов. При чтении пары (`Jonathan, John`) счетчики этих двух имен можно объединить. Конечно, необходимо запомнить, что эта пара встречалась, потому что в будущем может выясниться, что у `Jonathan` есть еще какой-нибудь эквивалент.

Воспользуемся хеш-таблицей (`L1`), связывающей имя с «истинным» именем. Также необходимо знать все имена, эквивалентные «истинному» имени. Эта информация будет храниться в хеш-таблице `L2`. Обратите внимание на то, что `L2` реализует обратный поиск по отношению к `L1`.

```
READ (Jonathan, John)
  L1.ADD Jonathan -> John
  L2.ADD John -> Jonathan
READ (Jon, Johnny)
  L1.ADD Jon -> Johnny
  L2.ADD Johnny -> Jon
READ (Johnny, John)
  L1.ADD Johnny -> John
  L1.UPDATE Jon -> John
  L2.UPDATE John -> Jonathan, Johnny, Jon
```

Если позднее обнаружится, что имя `John` эквивалентно, скажем, `Johny`, придется провести поиск по `L1` и `L2` и объединить все эквивалентные имена.

Такое решение работает, но ведение двух списков создает ненужные сложности.

Вместо этого можно рассматривать имена как «классы эквивалентности». Обнаружив пару (`Jonathan, John`), мы помещаем их в одно множество (или класс эквивалентности). Каждому имени ставится в соответствие определенный класс. Все имена в множестве соответствуют одному экземпляру множества.

Если нам понадобится объединить два множества, мы копируем одно множество в другое и обновляем хеш-таблицу указателем на новое множество.

```
READ (Jonathan, John)
  CREATE Set1 = Jonathan, John
  L1.ADD Jonathan -> Set1
  L1.ADD John -> Set1
READ (Jon, Johnny)
  CREATE Set2 = Jon, Johnny
  L1.ADD Jon -> Set2
  L1.ADD Johnny -> Set2
READ (Johnny, John)
  COPY Set2 into Set1.
    Set1 = Jonathan, John, Jon, Johnny
  L1.UPDATE Jon -> Set1
  L1.UPDATE Johnny -> Set1
```

На последнем шаге мы перебираем все элементы `Set2` и обновляем ссылку, чтобы она указывала на `Set1`. При этом отслеживается общее количество имен.

```
1  HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
2                                            String[][] synonyms) {
3    /* Разбор списка и инициализация классов эквивалентности.*/
4    HashMap<String, NameSet> groups = constructGroups(names);
5
6    /* Слияние классов эквивалентности. */
7    mergeClasses(groups, synonyms);
```

```
8
9    /* Обратное преобразование. */
10    return convertToMap(groups);
11  }
12
13  /* Основа алгоритма: прочитать каждую пару, объединить их классы
14   * эквивалентности и обновить отображения вторичного класса
15   * ссылкой на первое множество.*/
16  void mergeClasses(HashMap<String, NameSet> groups, String[][] synonyms) {
17    for (String[] entry : synonyms) {
18      String name1 = entry[0];
19      String name2 = entry[1];
20      NameSet set1 = groups.get(name1);
21      NameSet set2 = groups.get(name2);
22      if (set1 != set2) {
23        /* Меньшее множество всегда включается в большее. */
24        NameSet smaller = set2.size() < set1.size() ? set2 : set1;
25        NameSet bigger = set2.size() < set1.size() ? set1 : set2;
26
27        /* Слияние списков */
28        Set<String> otherNames = smaller.getNames();
29        int frequency = smaller.getFrequency();
30        bigger.copyNamesWithFrequency(otherNames, frequency);
31
32        /* Обновление отображений */
33        for (String name : otherNames) {
34          groups.put(name,  bigger);
35      }
36     }
37    }
38  }
39
40  /* Чтение пар (имя, счетчик) и инициализация отображения имен
41   * на коллекции NameSet (классы эквивалентности).*/
42  HashMap<String, NameSet> constructGroups(HashMap<String, Integer> names) {
43    HashMap<String, NameSet> groups = new HashMap<String, NameSet>();
44    for (Entry<String, Integer> entry : names.entrySet()) {
45      String name = entry.getKey();
46      int frequency = entry.getValue();
47      NameSet group = new NameSet(name, frequency);
48      groups.put(name,  group);
49    }
50    return groups;
51  }
52
53  HashMap<String, Integer> convertToMap(HashMap<String, NameSet> groups) {
54    HashMap<String, Integer> list = new HashMap<String, Integer>();
55    for (NameSet group : groups.values()) {
56      list.put(group.getRootName(), group.getFrequency());
57    }
58    return list;
59  }
60
61  public class NameSet {
62    private Set<String> names = new HashSet<String>();
63    private int frequency = 0;
64    private String rootName;
```

```
65
66    public NameSet(String name, int freq) {
67      names.add(name);
68      frequency = freq;
69      rootName = name;
70    }
71
72    public void copyNamesWithFrequency(Set<String> more, int freq) {
73      names.addAll(more);
74      frequency += freq;
75    }
76
77    public Set<String> getNames() { return names; }
78    public String getRootName() { return rootName; }
79    public int getFrequency() { return frequency; }
80    public int size() { return names.size(); }
81  }
```

Рассчитать время выполнения этого алгоритма не так просто. Чтобы получить искомую оценку, попробуем проанализировать худший случай.

Для этого алгоритма в худшем случае все имена эквивалентны, и нам приходится постоянно объединять множества. Кроме того, в худшем случае слияние осуществляется худшим из возможных способов: многократным попарным слиянием множеств. Каждое слияние требует копирования элементов множества в существующее множество и обновления указателей этих элементов. При большом размере множеств оно занимает значительную часть времени.

Если вы заметили аналогию с сортировкой слиянием (при которой одноэлементные массивы объединяются в двухэлементные, затем двухэлементные в четырехэлементные и т. д.), может возникнуть предположение, что алгоритм выполняется за время $O(N \log N)$. И это правильная оценка.

Если вы не заметили аналогию, можно пойти другим путем.

Представьте, что список состоит из имен (`a, b, c, d, ..., z`). В худшем случае элементы разбиваются на классы эквивалентности по парам: (`a, b`), (`c, d`), (`e, f`), ..., (`y, z`). Затем из полученных пар образуются новые: (`a, b, c, d`), (`e, f, g, h`), ..., (`w, x, y, z`) и т. д., пока не останется всего один класс.

На каждом проходе множества объединяются с перемещением половины элементов в новое множество. Каждый проход требует работы $O(N)$. (Множеств становится меньше, но сами множества увеличиваются.)

Сколько проходов будет сделано? При каждом проходе количество множеств сокращается вдвое. Следовательно, количество проходов составит $O(\log N)$. При $O(\log N)$ проходов и объеме работы на проход $O(N)$ общее время выполнения составит $O(N \log N)$.

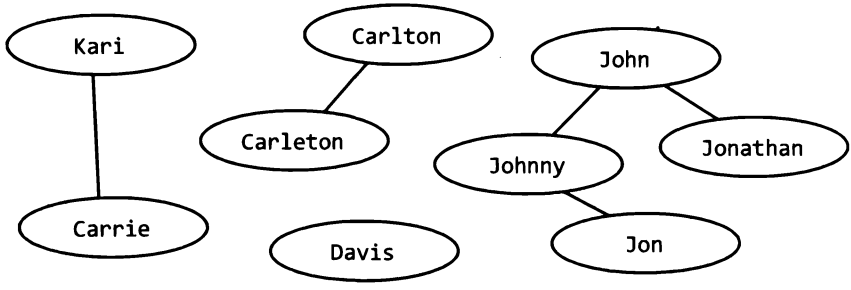
Это неплохой результат, но и его можно улучшить.

Оптимизированное решение

Чтобы оптимизировать старое решение, необходимо понять, что именно замедляет его. Ответ — слияние и обновление указателей.

Нельзя ли обойтись без этого? Допустим, как-то пометить существование отношения эквивалентности между двумя именами, но с самой информацией пока ничего не делать?

В сущности, речь идет о построении графа.



И что теперь? На рисунке все выглядит достаточно просто. Каждый компонент — множество эквивалентных имен. От нас потребуется лишь сгруппировать имена по компонентам, просуммировать счетчики и вернуть список со случайно выбранным именем из каждой группы.

Как это работает на практике? Можно выбрать имя и провести поиск в глубину (или ширину) для суммирования счетчиков всех имен одного компонента. Необходимо проследить за тем, чтобы каждый компонент обрабатывался только один раз. Делается это достаточно просто: узел, обнаруженный в ходе поиска по графу, помечается как посещенный, и поиск запускается только для тех узлов, для которых флаг посещения не установлен.

```
1  HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
2                                            String[][] synonyms) {
3    /* Создание данных. */
4    Graph graph = constructGraph(names);
5    connectEdges(graph, synonyms);
6
7    /* Поиск компонентов. */
8    HashMap<String, Integer> rootNames = getTrueFrequencies(graph);
9    return rootNames;
10 }
11
12 /* Добавление всех имен в качестве узлов графа. */
13 Graph constructGraph(HashMap<String, Integer> names) {
14   Graph graph = new Graph();
15   for (Entry<String, Integer> entry : names.entrySet()) {
16       String name = entry.getKey();
17       int frequency = entry.getValue();
18       graph.createNode(name, frequency);
19   }
20   return graph;
21 }
22
23 /* Соединение эквивалентных вариантов. */
24 void connectEdges(Graph graph, String[][] synonyms) {
25   for (String[] entry : synonyms) {
26       String name1 = entry[0];
27       String name2 = entry[1];
```

```
28        graph.addEdge(name1,  name2);
29    }
30  }
31
32  /* Выполнить поиск в глубину для каждого компонента. Если узел посещался
33   * ранее, его компонент уже обработан. */
34  HashMap<String, Integer> getTrueFrequencies(Graph graph) {
35    HashMap<String, Integer> rootNames = new HashMap<String, Integer>();
36    for (GraphNode node : graph.getNodes()) {
37      if (!node.isVisited()) { // Компонент уже посещался
38        int frequency = getComponentFrequency(node);
39        String name = node.getName();
40        rootNames.put(name, frequency);
41      }
42    }
43    return rootNames;
44  }
45
46  /* Провести поиск в глубину для нахождения суммарной частоты
47   * компонента и пометить каждый узел как посещенный.*/
48  int getComponentFrequency(GraphNode node) {
49    if (node.isVisited()) return 0; // Уже посещался
50
51    node.setIsVisited(true);
52    int sum = node.getFrequency();
53    for (GraphNode child : node.getNeighbors()) {
54      sum += getComponentFrequency(child);
55    }
56    return sum;
57  }
58
59  /* Код GraphNode и Graph особых пояснений не требует.
60   *  Он находится в архиве примеров.*/
```

Чтобы оценить общую эффективность, стоит проанализировать эффективность каждой части алгоритма.

- Чтение данных имеет линейную сложность относительно размера данных, поэтому оно выполняется за время $O(B + P)$, где B — количество имен, а P — количество пар синонимов. Это объясняется тем, что для каждого блока входных данных выполняется постоянный объем работы.
- Для вычисления частот каждое ребро «затрагивается» ровно один раз по всем вариантам поиска в графе, и каждый узел «затрагивается» ровно один раз для проверки того, посещался ли он ранее. Эта часть выполняется за время $O(B + P)$.

Следовательно, общее время алгоритма также составляет $O(B + P)$. И улучшить этот результат невозможно, потому что алгоритм должен как минимум прочитать $B + P$ блоков данных.

17.8. Цирк готовит новый номер — башню из людей, стоящих на плечах друг у друга. По практическим и эстетическим соображениям люди, стоящие выше, должны быть ниже ростом и легче, чем люди, находящиеся в основании башни. Известен вес и рост каждого акробата; напишите метод, вычисляющий наибольшее возможное число человек в башне.

Пример:
Ввод (ht, wt): (65, 100) (70, 150) (56, 90) (75, 190) (60, 95) (68, 110)
Вывод: максимальная высота башни — 6 человек, сверху вниз: (56, 90) (60, 95) (65, 100) (68, 110) (70, 150) (75, 190)

РЕШЕНИЕ

Если убрать из формулировки все лишнее, то задача сводится к следующему.

Есть список пар элементов. Нужно найти самую длинную последовательность элементов списка, такую, чтобы и первые и вторые элементы находились в «неубывающем» порядке.

Можно попробовать выполнить сортировку по атрибуту. Обычно это полезно, но в данном случае это лишь полдела.

Отсортировав элементы по высоте, мы получим относительный порядок, в котором дложны следовать элементы. Тем не менее мы еще должны найти самую длинную возрастающую подпоследовательность весов.

Решение 1. Рекурсивное

Одно из возможных решений — просто перебрать все возможности. После сортировки по высоте мы перебираем элементы массива. На каждом элементе открываются два пути: либо добавить элемент в подпоследовательность (если он действителен), либо не добавлять.

```
1  ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> items) {
2    Collections.sort(items);
3    return bestSeqAtIndex(items, new ArrayList<HtWt>(), 0);
4  }
5
6  ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array, ArrayList<HtWt> sequence,
7                                 int index) {
8    if (index >= array.size()) return sequence;
9
10   HtWt value = array.get(index);
11
12   ArrayList<HtWt> bestWith = null;
13   if (canAppend(sequence, value)) {
14     ArrayList<HtWt> sequenceWith = (ArrayList<HtWt>) sequence.clone();
15     sequenceWith.add(value);
16     bestWith = bestSeqAtIndex(array, sequenceWith, index + 1);
17   }
18
19   ArrayList<HtWt> bestWithout = bestSeqAtIndex(array, sequence, index + 1);
20
21   if (bestWith == null || bestWithout.size() > bestWith.size()) {
22     return bestWithout;
23   } else {
24     return bestWith;
25   }
26 }
27
28 boolean canAppend(ArrayList<HtWt> solution, HtWt value) {
29   if (solution == null) return false;
```

```
30    if (solution.size() == 0) return true;
31
32    HtWt last = solution.get(solution.size() - 1);
33    return last.isBefore(value);
34  }
35
36  ArrayList<HtWt> max(ArrayList<HtWt> seq1, ArrayList<HtWt> seq2) {
37    if (seq1 == null) {
38      return seq2;
39    } else if (seq2 == null) {
40      return seq1;
41    }
42    return seq1.size() > seq2.size() ? seq1 : seq2;
43  }
44
45  public class HtWt implements Comparable<HtWt> {
46    private int height;
47    private int weight;
48    public HtWt(int h, int w) { height = h; weight = w; }
49
50    public int compareTo(HtWt second) {
51      if (this.height != second.height) {
52        return ((Integer)this.height).compareTo(second.height);
53      } else {
54        return ((Integer)this.weight).compareTo(second.weight);
55     }
56    }
57
58    /* Возвращает true, если элемент "this" должен предшествовать "other".
59     * Учтите, что this.isBefore(other) и other.isBefore(this) могут быть
60     * ложными одновременно (этим метод отличается от compareTo) */
61    public boolean isBefore(HtWt other) {
62      if (height < other.height && weight < other.weight) {
63        return true;
64      } else {
65        return false;
66     }
67    }
68  }
```

Алгоритм выполняется за время $O(2^n)$. Его можно оптимизировать за счет применения мемоизации (то есть кэширования лучших последовательностей), но существует другое, более элегантное решение.

Решение 2. Итеративное

Допустим, известна самая длинная подпоследовательность, завершаемая каждым из элементов от `A[0]` до `A[3]`. Можно ли использовать эту информацию для нахождения самой длинной подпоследовательности, завершающейся элементом `A[4]`?

```
Массив: 13, 14, 10, 11, 12
Самая длинная подпоследовательность (завершаемая A[0]): 13
Самая длинная подпоследовательность (завершаемая A[1]): 13, 14
Самая длинная подпоследовательность (завершаемая A[2]): 10
Самая длинная подпоследовательность (завершаемая A[3]): 10, 11
Самая длинная подпоследовательность (завершаемая [4]): 10, 11, 12
```

Конечно. Нужно просто присоединить элемент A[4] к самой длинной подпоследовательности, к которой он может быть присоединен. Этот алгоритм реализуется достаточно тривиально.

```
1  ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> array) {
2    Collections.sort(array);
3
4    ArrayList<ArrayList<HtWt>> solutions =  new ArrayList<ArrayList<HtWt>>();
5    ArrayList<HtWt> bestSequence = null;
6
7    /* Поиск самой длинной подпоследовательности, завершающейся
8     * на каждом элементе. */
9    for (int i = 0; i < array.size(); i++) {
10     ArrayList<HtWt> longestAtIndex = bestSeqAtIndex(array, solutions, i);
11     solutions.add(i, longestAtIndex);
12     bestSequence = max(bestSequence, longestAtIndex);
13   }
14
15   return bestSequence;
16 }
17
18 /* Поиск самой длинной подпоследовательности. */
19 ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array,
20     ArrayList<ArrayList<HtWt>> solutions, int index) {
21   HtWt value = array.get(index);
22
23   ArrayList<HtWt> bestSequence = new ArrayList<HtWt>();
24   /* Поиск самой длинной подпоследовательности, к которой
25    * можно присоединить элемент. */
26   for (int i = 0; i < index; i++) {
27     ArrayList<HtWt> solution = solutions.get(i);
28     if (canAppend(solution, value)) {
29       bestSequence = max(solution, bestSequence);
30     }
31   }
32
33   /* Присоединение элемента. */
34   ArrayList<HtWt> best = (ArrayList<HtWt>) bestSequence.clone();
35   best.add(value);
36
37   return best;
38 }
```

Этот алгоритм требует $O(n^2)$ времени. Алгоритм с временем $O(n \log(n))$ существует, но он значительно сложнее, и мало вероятно, что вы построите его на собеседовании, даже с помощью интервьюера. Однако если вы заинтересовались данным вопросом, информацию можно легко найти в Интернете.

17.9. Разработайте алгоритм, позволяющий найти k-е число, простыми множителями которого являются только числа 3, 5 и 7. Обратите внимание: 3, 5 и 7 не обязаны быть множителями, но других простых множителей быть не должно. Например, несколько первых таких чисел — 1, 3, 5, 7, 9, 15, 21.

РЕШЕНИЕ

Для начала нужно понять, что же, собственно, нужно найти. По условию задачи ищется k-е число вида $3^a * 5^b * 7^c$. Начнем с поиска решения методом «грубой силы».

Метод «грубой силы»

Мы знаем, что максимальное значение k-го числа ограничивается $3^k * 5^k * 7^k$. Таким образом, в «тупом» решении вычисляются значения $3^a * 5^b * 7^c$ для всех a, b и c в диапазоне от 0 до k. Далее мы заносим их в список, сортируем список и выбираем k-е значение по возрастанию.

```
1  int getKthMagicNumber(int k) {
2    ArrayList<Integer> possibilities = allPossibleKFactors(k);
3    Collections.sort(possibilities);
4    return possibilities.get(k);
5  }
6
7  ArrayList<Integer> allPossibleKFactors(int k) {
8    ArrayList<Integer> values = new ArrayList<Integer>();
9    for (int a = 0; a <= k; a++) { // Цикл для 3
10     int powA = (int) Math.pow(3,  a);
11     for (int b = 0; b <= k; b++) { // Цикл для 5
12       int powB = (int) Math.pow(5,  b);
13       for (int c = 0; c <= k; c++) { // Цикл для 7
14         int powC = (int) Math.pow(7, c);
15         int value = powA * powB * powC;
16
17         /* Проверка переполнения. */
18         if (value < 0 || powA == Integer.MAX_VALUE ||
19            powB == Integer.MAX_VALUE ||
20            powC == Integer.MAX_VALUE) {
21           value = Integer.MAX_VALUE;
22       }
23         values.add(value);
24     }
25    }
26   }
27   return values;
28 }
```

За какое время будет выполняться это решение? В нем используются вложенные циклы for, каждый из которых состоит из k итераций. Время выполнения всех `allPossibleKFactors` составляет $O(k^3)$. Затем k^3 результатов сортируются за время $O(k^3 \log(k^3))$, что эквивалентно $O(k^3 \log k)$.

Существует ряд возможных оптимизаций (и более эффективных способов обработки целочисленного переполнения), но откровенно говоря, алгоритм достаточно медленный. Лучше сосредоточиться не на оптимизации, а на переработке алгоритма.

Усовершенствованный алгоритм

Представим, как должен выглядеть результат.

1	-	$3^0 * 5^0 * 7^0$
3	3	$3^1 * 5^0 * 7^0$
5	5	$3^0 * 5^1 * 7^0$
7	7	$3^0 * 5^0 * 7^1$
9	3*3	$3^2 * 5^0 * 7^0$
15	3*5	$3^1 * 5^1 * 7^0$
21	3*7	$3^1 * 5^0 * 7^1$
25	5*5	$3^0 * 5^2 * 7^0$
27	3*9	$3^3 * 5^0 * 7^0$
35	5*7	$3^0 * 5^1 * 7^1$
1	-	$3^0 * 5^0 * 7^0$
45	5*9	$3^2 * 5^1 * 7^0$
49	7*7	$3^0 * 5^0 * 7^2$
63	3*21	$3^2 * 5^0 * 7^1$

Вопрос: как должно выглядеть следующее значение в списке? Возможны три варианта:

- `3 * (некоторое предыдущее число из числового ряда)`;
- `5 * (некоторое предыдущее число из числового ряда)`;
- `7 * (некоторое предыдущее число из числового ряда)`.

Если вам этот факт не кажется очевидным, взгляните на происходящее так: каким бы ни было следующее значение (обозначим его `nv`), разделим его на 3. Результат уже присутствует в списке? Раз `nv` кратно 3 — да. То же самое можно сказать о делении на 5 и 7.

Итак, мы знаем, что A_k можно записать как `(3, 5 или 7) * (некоторое значение из` $\{A_1, \ldots, A_{k-1}\}$). Также известно, что по определению A_k является следующим числом в данном ряду. Поэтому A_k должно быть наименьшим «новым» числом (числом, уже присутствующим в $\{A_1, \ldots, A_{k-1}\}$), которое может быть получено умножением каждого значения в списке на 3, 5 или 7.

Как найти A_k? Можно умножить каждое число в списке на 3, 5 или 7 и найти наименьший элемент, который еще не был добавлен в список. Решение потребует времени $O(k^2)$. Неплохо, но можно сделать и лучше.

Вместо попыток «вывести» A_k из предыдущих элементов списка (умножением на 3, 5 или 7) можно рассматривать каждое предыдущее значение как основу для расчета трех последующих значений. Таким образом, каждое число A_i может использоваться для формирования следующих значений:

$3 * A_i$
$5 * A_i$
$7 * A_i$

Это обстоятельство поможет заложить основу для последующего планирования. Каждый раз, когда в список добавляется число A_i, значения $3A_i$, $5A_i$ и $7A_i$ заносятся в «резервный» список. Чтобы сгенерировать A_{i+1}, достаточно будет найти наименьшее значение во временном списке.

Код может выглядеть так:

```
1  int removeMin(Queue<Integer> q) {
2    int min = q.peek();
3    for (Integer v : q) {
4      if (min > v) {
5        min = v;
6      }
7    }
8    while (q.contains(min)) {
9      q.remove(min);
10   }
11   return min;
12 }
13
14 void addProducts(Queue<Integer> q, int v) {
15   q.add(v * 3);
16   q.add(v * 5);
17   q.add(v * 7);
18 }
19
20 int getKthMagicNumber(int k) {
21   if (k < 0) return 0;
22
23   int val = 1;
24   Queue<Integer> q = new LinkedList<Integer>();
25   addProducts(q, 1);
26   for (int i = 0; i < k; i++) {
27     val = removeMin(q);
28     addProducts(q, val);
29   }
30   return val;
31 }
```

Данный алгоритм намного лучше предыдущего, но он все еще не идеален.

Оптимальный алгоритм

Чтобы сгенерировать новый элемент A_i, мы проводим поиск по связному списку, где каждый элемент имеет вид:

- 3 * предыдущий элемент;
- 5 * предыдущий элемент;
- 7 * предыдущий элемент.

Где та лишняя работа, которую можно было бы устранить в результате оптимизации?

Допустим, список имеет вид:

$q_6 = \{7A_1, 5A_2, 7A_2, 7A_3, 3A_4, 5A_4, 7A_4, 5A_5, 7A_5\}$

Когда мы ищем минимальный элемент в списке, то проверяем сначала $7A_1$ < min, а затем $7A_5$ < min. Неразумно, не правда ли? Поскольку мы знаем, что $A_1 < A_5$, то достаточно выполнить проверку $7A_i$ < min.

Если бы список был с самого начала разделен по постоянным множителям, то было бы достаточно проверить только первое из значений, кратных 3, 5 и 7. Все последующие элементы будут больше.

Таким образом, список принимает вид:

```
Q36 = {3A4}
Q56 = {5A2, 5A4, 5A5}
Q76 = {7A1, 7A2, 7A3, 7A4, 7A5}
```

Чтобы найти минимум, достаточно проверить начало каждой очереди:

```
y = min(Q3.head(), Q5.head(), Q7.head())
```

Сразу же после вычисления `y` нужно добавить `3y` в список `Q3`, `5y` в `Q5` и `7y` в `Q7`. Но эти элементы должны вставляться только в том случае, если они отсутствуют в других списках.

Как, например, `3y` может попасть в какой-нибудь другой список? Допустим, элемент `y` был получен из `Q7`, это означает, что `y = 7x` для некоторого меньшего `x`. Если `7x` — наименьшее значение, значит, `3x` уже было задействовано. А как мы действовали при обнаружении `3x`? Вставили `7 * 3x` в `Q7`. Обратите внимание, что `7 * 3x = 3 * * 7x = 3y`.

Объясним иначе: если мы берем элемент из `Q7`, он будет иметь вид `7 * suffix`. Мы знаем, что `3 * suffix` и `5 * suffix` уже обработаны, и элемент `7 * 3 * suffix` добавлен в `Q7`. При обработке `5 * suffix` мы знаем, что `7 * 5 * suffix` был добавлен в `Q7`. Пока еще не встречалось только значение `7 * 7 * suffix`, поэтому мы добавляем его в `Q7`.

Рассмотрим конкретный пример, чтобы все стало совершенно ясно:

```
инициализация:

    Q3 = 3
    Q5 = 5
    Q7 = 7
удаляем min = 3. вставляем 3*3 в Q3, 5*3 в Q5, 7*3 в Q7
    Q3 = 3*3
    Q5 = 5, 5*3
    Q7 = 7, 7*3
удаляем min = 5. 3*5 - дубль, значит, мы уже обработали 5*3.
Вставляем 5*5 в Q5, 7*5 в Q7
    Q3 = 3*3
    Q5 = 5*3, 5*5
    Q7 = 7, 7*3, 7*5.
удаляем min = 7. 3*7 и 5*7 - дубли, уже обработали 7*3 и 7*5. Вставляем 7*7 в Q7
    Q3 = 3*3
    Q5 = 5*3, 5*5
    Q7 = 7*3, 7*5, 7*7
удаляем min = 3*3 = 9. вставляем 3*3*3 в Q3, 3*3*5 в Q5, 3*3*7 в Q7.
    Q3 = 3*3*3
    Q5 = 5*3, 5*5, 5*3*3
    Q7 = 7*3, 7*5, 7*7, 7*3*3
удаляем min = 5*3 = 15. 3*(5*3) - дубль, так как уже обработали 5*(3*3).
Вставляем 5*5*3 в Q5, 7*5*3 в Q7
    Q3 = 3*3*3
    Q5 = 5*5, 5*3*3, 5*5*3
    Q7 = 7*3, 7*5, 7*7, 7*3*3, 7*5*3
удаляем min = 7*3 = 21. 3*(7*3) и 5*(7*3) - дубли,
уже обработали 7*(3*3) и 7*(5*3). Вставляем 7*7*3 в Q7
```

```
Q3 = 3*3*3
Q5 = 5*5, 5*3*3, 5*5*3
Q7 = 7*5, 7*7, 7*3*3, 7*5*3, 7*7*3
```

На псевдокоде этот алгоритм имеет следующий вид:

1. Инициализировать array и очереди Q3, Q5 и Q7.
2. Вставить 1 в array.
3. Вставить 1*3, 1*5 и 1*7 в Q3, Q5 и Q7 соответственно.
4. Пусть x — минимальный элемент в Q3, Q5 и Q7. Присоединить x к magic.
5. Если x находится в:

   Q3 -> присоединить x*3, x*5 и x*7 к Q3, Q5 и Q7. Удалить x из Q3.

   Q5 -> присоединить x*5 и x*7 к Q5 и Q7. Удалить x из Q5.

   Q7 -> присоединить x*7 только к Q7. Удалить x из Q7.

6. Повторить шаги 4–6, пока k-й элемент не будет найден.

Ниже приведена реализация этого алгоритма.

```
1  int getKthMagicNumber(int k) {
2    if (k < 0) {
3      return 0;
4    }
5    int val = 0;
6    Queue<Integer> queue3 = new LinkedList<Integer>();
7    Queue<Integer> queue5 = new LinkedList<Integer>();
8    Queue<Integer> queue7 = new LinkedList<Integer>();
9    queue3.add(1);
10
11    /* итерации с 0-й по k-ю. */
12    for (int i = 0; i <= k; i++) {
13      int v3 = queue3.size() > 0 ? queue3.peek() : Integer.MAX_VALUE;
14      int v5 = queue5.size() > 0 ? queue5.peek() : Integer.MAX_VALUE;
15      int v7 = queue7.size() > 0 ? queue7.peek() : Integer.MAX_VALUE;
16      val = Math.min(v3, Math.min(v5, v7));
17      if (val == v3) { // В очереди 3, 5 и 7
18        queue3.remove();
19        queue3.add(3 * val);
20        queue5.add(5 * val);
21      } else if (val == v5) { // В очереди 5 и 7
22        queue5.remove();
23        queue5.add(5 * val);
24      } else if (val == v7) { // В очередь Q7
25        queue7.remove();
26     }
27      queue7.add(7 * val); // Всегда в Q7
28    }
29    return val;
30  }
```

Если вам досталась подобная задача, приложите все усилия, чтобы ее решить, — при том, что она действительно трудна. Можно начать с решения методом «грубой силы» (спорно, зато не слишком сложно), а затем попытаться оптимизировать полученное решение. Или попытайтесь найти закономерность в числах.

Если у вас возникнут затруднения, скорее всего, интервьюер поможет. Что бы ни было, не сдавайтесь! Рассуждайте вслух, задавайте вопросы и объясняйте ход ваших мыслей. Интервьюер наверняка начнет помогать вам.

Помните: никто не ожидает, что вы найдете идеальное решение. Ваши результаты будут сравниваться с результатами других кандидатов. Сложные задачи сложны для всех.

17.10. «Доминирующим значением» называется значение, присутствующее более чем в половине элементов массива. Для заданного массива с положительными целыми числами найдите доминирующее значение. Если доминирующего значения не существует, верните –1. Поиск должен быть выполнен за время $O(N)$ и с затратами памяти $O(1)$.

Пример:

Ввод: `1 2 5 9 5 9 5 5 5`

Вывод: `5`

РЕШЕНИЕ

Рассмотрим пример:

```
3 1 7 1 3 7 3 7 1 7 7
```

Заметим, что если доминирующий элемент (7 в данном случае) редко встречается в начале, то ближе к концу он начинает встречаться намного чаще. Это полезное наблюдение.

В этой задаче указаны конкретные параметры сложности решения: время $O(N)$, затраты памяти $O(1)$. Тем не менее будет полезно снять одно из этих ограничений и попытаться разработать алгоритм. Попробуем снять ограничения по времени, но сохранить пространственную сложность $O(1)$.

Решение 1 (медленное)

В этом варианте мы просто перебираем массив и проверяем каждый элемент на то, является ли он доминирующим. Проверка выполняется за время $O(N^2)$ при затратах памяти $O(1)$.

```
1  int findMajorityElement(int[] array) {
2    for (int x : array) {
3      if (validate(array, x)) {
4        return x;
5      }
6    }
7    return -1;
8  }
9
10 boolean validate(int[] array, int majority) {
11   int count = 0;
12   for (int n : array) {
13     if (n == majority) {
14       count++;
15     }
16   }
```

```
17
18    return count > array.length / 2;
19  }
```

Это решение не удовлетворяет временным ограничениям задачи, но может стать отправной точкой для дальнейшего анализа. Подумаем, как его можно оптимизировать.

Решение 2 (оптимальное)

Давайте проанализируем работу алгоритма в этом конкретном примере. Выполняется ли здесь какая-либо лишняя работа?

3	1	7	1	1	7	7	3	7	7	7
0	1	2	3	4	5	6	7	8	9	10

На первом проходе выбирается элемент 3, который должен проверяться на роль доминирующего элемента. Через несколько позиций снова остается всего одно вхождение 3 с несколькими другими элементами. Нужно ли продолжать проверку 3?

С одной стороны — да. Элемент 3 еще может «реабилитироваться» и стать доминирующим, если массив завершается группой «троек».

С другой стороны — не обязательно. Если 3 «реабилитируется», то все эти «тройки» будут обнаружены позднее, на последующем шаге проверки. Текущий шаг `validate(3)` можно завершить.

Для первого элемента такая логика подходит, но как насчет следующего элемента? Мы будем немедленно завершать `validate(1)`, `validate(7)` и т. д.

Раз эта логика нормально работала для первого элемента, нельзя ли рассматривать каждый последующий элемент как первый элемент некоторого подмассива? Это будет означать, что `validate(array[1])` будет начинать проверку с индекса 1, `validate(array[2])` — с индекса 2, и т. д.

Как это будет выглядеть?

```
validate(3)
  видит 3 -> countYes = 1, countNo = 0
  видит 1 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 3 пока не является доминирующим элементом.
validate(1)
  видит 1 -> countYes = 0, countNo = 0
  видит 7 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 1 пока не является доминирующим элементом.
validate(7)
  видит 7 -> countYes = 1, countNo = 0
  видит 1 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 7 пока не является доминирующим элементом.
validate(1)
  видит 1 -> countYes = 1, countNo = 0
  видит 1 -> countYes = 2, countNo = 0
  видит 7 -> countYes = 2, countNo = 1
  видит 7 -> countYes = 2, countNo = 1
  ЗАВЕРШЕНИЕ. 1 пока не является доминирующим элементом.
validate(1)
  видит 1 -> countYes = 1, countNo = 0
  видит 7 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 1 пока не является доминирующим элементом.
```

```
validate(7)
  видит 7 -> countYes = 1, countNo = 0
  видит 7 -> countYes = 2, countNo = 0
  видит 3 -> countYes = 2, countNo = 1
  видит 7 -> countYes = 3, countNo = 1
  видит 7 -> countYes = 4, countNo = 1
  видит 7 -> countYes = 5, countNo = 1
```

Знаем ли мы на этой стадии, что 7 является доминирующим элементом? Не обязательно. Мы исключили все, что предшествует 7, и все, что идет после. Но ведь доминирующего элемента может и не быть! Быстрая проверка `validate(7)`, начинающаяся с самого начала, может подтвердить, что 7 действительно является доминирующим элементом. Проверка потребует времени $O(N)$, а следовательно, не повлияет на общее время выполнения.

Результат неплохой, но посмотрим, нельзя ли немного улучшить его. Заметим, что некоторые элементы «проверяются» многократно. Можно ли избавиться от этих проверок?

Рассмотрим первый вызов `validate(3)`. Он завершается неудачей после подмассива `[3, 1]`, потому что 3 не является доминирующим элементом. Но из этого также следует, что в этом подмассиве нет других доминирующих элементов. По логике, описанной выше, вызов `validate(1)` оказывается лишним — известно, что 1 не встречается более чем в половине случаев. Если это доминирующий элемент, он проявится позднее.

Опробуем эту схему и посмотрим, работает ли она.

```
validate(3)
  видит 3 -> countYes = 1, countNo = 0
  видит 1 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 3 пока не является доминирующим элементом.
пропустить 1
validate(7)
  видит 7 -> countYes = 1, countNo = 0
  видит 1 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 7 пока не является доминирующим элементом.
пропустить 1
validate(1)
  видит 1 -> countYes = 1, countNo = 0
  видит 7 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 1 пока не является доминирующим элементом.
пропустить 7
validate(7)
  видит 7 -> countYes = 1, countNo = 0
  видит 3 -> countYes = 1, countNo = 1
  ЗАВЕРШЕНИЕ. 7 пока не является доминирующим элементом.
пропустить 3
validate(7)
  видит 7 -> countYes = 1, countNo = 0
  видит 7 -> countYes = 2, countNo = 0
  видит 7 -> countYes = 3, countNo = 0
```

Хорошо! Мы получили правильный ответ. А может, просто повезло?

Задумаемся на минуту над тем, что делает этот алгоритм.

1. Мы начинаем с [3] и расширяем подмассив до тех пор, пока 3 не перестает быть доминирующим элементом. Это происходит в состоянии [3, 1]. На момент завершения в подмассиве нет доминирующего элемента.
2. Затем мы переходим к [7] и расширяем подмассив до [7, 1]. И снова проверка прекращается, а в подмассиве доминирующего элемента быть не может.
3. Переходим к [1] и расширяем подмассив до [1, 7]. Завершение, доминирующего элемента быть не может.
4. Переходим к [7] и расширяем подмассив до [7, 3]. Завершение, доминирующего элемента быть не может.
5. Переходим к [7] и расширяем подмассив до конца массива: [7, 7, 7]. Доминирующий элемент найден (и теперь его необходимо проверить).

При каждом завершении шага `validate` в подмассиве нет доминирующего элемента. Это означает, что количество элементов, отличных от 7, по крайней мере не меньше количества 7. И хотя подмассив фактически удаляется из исходного массива, доминирующий элемент может быть найден в оставшейся части массива — и по-прежнему будет обладать статусом доминирования. А это означает, что в какой-то момент доминирующий элемент будет обнаружен.

Теперь алгоритм может выполняться за два прохода: на первом проходе ищется потенциальный доминирующий элемент, а на втором он проверяется. Вместо двух разных переменных (`countYes` и `countNo`) используется одна переменная `count`, значение которой увеличивается и уменьшается по мере необходимости.

```
1  int findMajorityElement(int[] array) {
2    int candidate = getCandidate(array);
3    return validate(array, candidate) ? candidate : -1;
4  }
5
6  int getCandidate(int[] array) {
7    int majority = 0;
8    int count = 0;
9    for (int n : array) {
10     if (count == 0) { // Нет доминирующего элемента.
11       majority = n;
12     }
13     if (n == majority) {
14       count++;
15     } else {
16       count--;
17     }
18   }
19   return majority;
20 }
21
22 boolean validate(int[] array, int majority) {
23   int count = 0;
24   for (int n : array) {
25     if (n == majority) {
26       count++;
27     }
28   }
29
```

```
30    return count > array.length / 2;
31  }
```

Алгоритм выполняется за время $O(N)$ с затратами памяти $O(1)$.

17.11. Имеется большой текстовый файл, содержащий слова. Напишите код, который позволяет найти минимальное расстояние в файле (выражаемое количеством слов) между двумя заданными словами. Если операция будет многократно выполняться для одного файла (но с разными парами слов), можно ли оптимизировать решение?

РЕШЕНИЕ

Будем считать, что порядок появления слов `word1` и `word2` не важен. Этот вопрос нужно согласовать с интервьюером.

Чтобы решить эту задачу, достаточно будет прочитать файл только один раз. При этом мы сохраним информацию о том, где находились последние встреченные `word1` или `word2`, в `location1` и `location2` соответственно. Если текущее расстояние лучше хранимого, мы обновляем его.

Приведенный далее код иллюстрирует этот алгоритм:

```
1  LocationPair findClosest(String[] words, String word1, String word2) {
2    LocationPair best = new LocationPair(-1, -1);
3    LocationPair current = new LocationPair(-1, -1);
4    for (int i = 0; i < words.length; i++) {
5      String word = words[i];
6      if (word.equals(word1)) {
7        current.location1 = i;
8        best.updateWithMin(current);
9      } else if (word.equals(word2)) {
10       current.location2 = i;
11       best.updateWithMin(current); // Если расстояние меньше, обновить
12     }
13   }
14   return best;
15 }
16
17 public class LocationPair {
18   public int location1, location2;
19   public LocationPair(int first, int second) {
20     setLocations(first, second);
21   }
22
23   public void setLocations(int first, int second) {
24     this.location1 = first;
25     this.location2 = second;
26   }
27
28   public void setLocations(LocationPair loc) {
29     setLocations(loc.location1, loc.location2);
30   }
31
32   public int distance() {
33     return Math.abs(location1 - location2);
```

```
34    }
35
36    public boolean isValid() {
37      return location1 >= 0 && location2 >= 0;
38    }
39
40    public void updateWithMin(LocationPair loc) {
41      if (!isValid() || loc.distance() < distance()) {
42        setLocations(loc);
43      }
44    }
45  }
```

Если ту же работу потребуется выполнить для других пар слов, можно создать хеш-таблицу, связывающую слова с позицией в файле. Тогда будет достаточно прочитать весь список слов всего один раз. После этого можно выполнить очень похожий алгоритм, но только перебрать позиции напрямую.

Рассмотрим следующие списки:

```
listA: {1, 2, 9, 15, 25}
listB: {4, 10, 19}
```

Представьте указатели pA и pB, указывающие на начало каждого списка. Наша цель — добиться того, чтобы pA и pB указывали на значения, как можно более близкие.

Первая потенциальная пара — (1, 4).

Какую первую пару мы сможем найти? Если переместить pB, то расстояние определенно увеличится. Но если переместить pA, мы можем получить улучшенную пару. Так и поступим.

Вторая потенциальная пара — (2, 4). Она лучше предыдущей, поэтому мы сохраним ее как лучшую пару.

Мы снова перемещаем pA и получаем (9, 4). Эта пара хуже предыдущей.

Теперь, поскольку значение элемента pA больше значения элемента pB, мы перемещаем pB и получаем (9, 10). Затем будут получены пары (15, 10), (15, 19) и (25, 19).

Ниже приведена реализация этого алгоритма.

```
1  LocationPair findClosest(String word1, String word2,
2                           HashMapList<String, Integer> locations) {
3    ArrayList<Integer> locations1 = locations.get(word1);
4    ArrayList<Integer> locations2 = locations.get(word2);
5    return findMinDistancePair(locations1, locations2);
6  }
7
8  LocationPair findMinDistancePair(ArrayList<Integer> array1,
9                                   ArrayList<Integer> array2) {
10   if (array1 == null || array2 == null || array1.size() == 0 ||
11       array2.size() == 0) {
12     return null;
13   }
14
15   int index1 = 0;
16   int index2 = 0;
```

```
17    LocationPair best = new LocationPair(array1.get(0), array2.get(0));
18    LocationPair current = new LocationPair(array1.get(0), array2.get(0));
19
20    while (index1 < array1.size() && index2 < array2.size()) {
21      current.setLocations(array1.get(index1), array2.get(index2));
22      best.updateWithMin(current); // Если расстояние меньше, обновить
23      if (current.location1 < current.location2) {
24        index1++;
25      } else {
26        index2++;
27      }
28    }
29
30    return best;
31  }
32
33  /* Предварительная обработка. */
34  HashMapList<String, Integer> getWordLocations(String[] words) {
35    HashMapList<String, Integer> locations = new HashMapList<String, Integer>();
36    for (int i = 0; i < words.length; i++) {
37      locations.put(words[i], i);
38    }
39    return locations;
40  }
41
42  /* HashMapList<String, Integer> связывает String
43   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Предварительная обработка в этом алгоритме выполняется за время $O(N)$, где N — количество слов в строке.

Поиск ближайшей пары занимает время $O(A + B)$, где A — количество вхождений первого слова, а B — количество вхождений второго слова.

17.12. Рассмотрим простую структуру данных BiNode, содержащую указатели на два других узла:

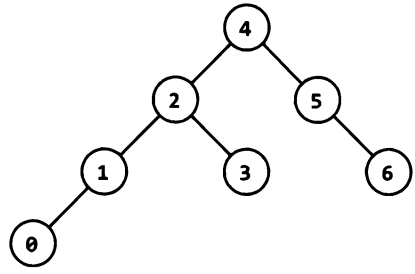
```
public class BiNode {
    public BiNode node1, node2;
    public int data;
}
```

Структура данных BiNode может использоваться для представления бинарного дерева (где node1 — левый узел и node2 — правый узел) или двунаправленного связного списка (где node1 — предыдущий узел, а node2 — следующий узел). Реализуйте метод, преобразующий бинарное дерево поиска (реализованное с помощью BiNode) в двунаправленный связный список. Значения должны храниться упорядоченно, а операция должна выполняться «на месте» (то есть прямо в исходной структуре данных).

РЕШЕНИЕ

Эта сложная на первый взгляд задача элегантно решается с помощью рекурсии. Впрочем, для ее решения необходимо очень хорошо понимать механизмы рекурсии.

Рассмотрим простое бинарное дерево поиска:



Метод `convert` преобразует его в двусвязный список:

```
0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6
```

Давайте рекурсивно обойдем его, начиная с корня (узел 4).

Мы знаем, что левые и правые половины дерева формируют собственные «подразделения» связного списка (то есть располагаются в связном списке последовательно). Если преобразовать левые и правые поддеревья в двусвязный список, сможем ли мы создать итоговый связный список?

Да! Нужно просто объединить части.

Псевдокод выглядит примерно так:

```
1  BiNode convert(BiNode node) {
2    BiNode left = convert(node.left);
3    BiNode right = convert(node.right);
4    mergeLists(left, node, right);
5    return left; // Начало left
6  }
```

Чтобы реализовать основные элементы этого решения, необходимо получить доступ к началу и концу каждого связанного списка. Это можно сделать несколькими способами.

Решение 1. Дополнительная структура данных

Первое (более простое) решение — создание новой структуры данных `NodePair`, которая будет хранить только начало и конец связного списка. Метод `convert` сможет возвратить результат типа `NodePair`.

Приведенный далее код реализует данный подход:

```
1  private class NodePair {
2    BiNode head, tail;
3
4    public NodePair(BiNode head, BiNode tail)  {
5      this.head = head;
6      this.tail = tail;
7    }
8  }
9
10 public NodePair convert(BiNode root) {
11   if (root == null) return null;
12
```

```
13    NodePair part1 = convert(root.node1);
14    NodePair part2 = convert(root.node2);
15
16    if (part1 != null) {
17      concat(part1.tail, root);
18    }
19
20    if (part2 != null) {
21      concat(root, part2.head);
22    }
23
24    return new NodePair(part1 == null ? root : part1.head,
25                        part2 == null ? root : part2.tail);
26  }
27
28  public static void concat(BiNode x, BiNode y) {
29    x.node2 = y;
30    y.node1 = x;
31  }
```

Этот код по-прежнему преобразует структуру данных `BiNode` на месте. Класс `NodePair` используется только для возврата дополнительных данных. С таким же успехом можно использовать двухэлементный массив `BiNode`, но это будет более «неряшливое» решение (а все мы любим «чистый» код, особенно на собеседовании).

Тем не менее было бы хорошо обойтись без этих дополнительных структур данных, и это возможно.

Решение 2. Получение конечного элемента

Вместо возврата начала и конца связного списка в `NodePair` можно возвращать только начало, а затем использовать его для нахождения конечного узла:

```
1  BiNode convert(BiNode root) {
2    if (root == null) return null;
3
4    BiNode part1 = convert(root.node1);
5    BiNode part2 = convert(root.node2);
6
7    if (part1 != null) {
8      concat(getTail(part1), root);
9    }
10
11    if (part2 != null) {
12      concat(root, part2);
13    }
14
15    return part1 == null ? root : part1;
16  }
17
18  public static BiNode getTail(BiNode node) {
19    if (node == null) return null;
20    while (node.node2 != null) {
21      node = node.node2;
22    }
23    return node;
24  }
```

Не считая вызова `getTail`, этот код практически совпадает с предыдущим решением. Это не очень эффективно. `getTail` будет `d` раз обрабатывать листовой узел на глубине `d` (по одному разу для каждого вышестоящего узла), в результате общее время выполнения составит $O(N^2)$, где N — количество узлов в дереве.

Решение 3. Создание кольцевого связного списка

Третье и последнее решение строится на базе второго.

Для него потребуется вернуть начало и конец связного списка в `BiNode`. Впрочем, для этого можно просто вернуть начало *циклического* связного списка. Чтобы получить конечный узел, достаточно обратиться к `head.node1`.

```
1  BiNode convertToCircular(BiNode root) {
2    if (root == null) return null;
3
4    BiNode part1 = convertToCircular(root.node1);
5    BiNode part3 = convertToCircular(root.node2);
6
7    if (part1 == null && part3 == null) {
8      root.node1 = root;
9      root.node2 = root;
10     return root;
11   }
12   BiNode tail3 = (part3 == null) ? null : part3.node1;
13
14   /* Соединение левой части с корнем */
15   if (part1 == null) {
16     concat(part3.node1, root);
17   } else {
18     concat(part1.node1, root);
19   }
20
21   /* Соединение правой части с корнем */
22   if (part3 == null) {
23     concat(root, part1);
24   } else {
25     concat(root, part3);
26   }
27
28   /* Соединение правой части с левой */
29   if (part1 != null && part3 != null) {
30     concat(tail3, part1);
31   }
32
33   return part1 == null ? root : part1;
34 }
35
36 /* Преобразовать в циклический связный список и разорвать цикл. */
37 BiNode convert(BiNode root) {
38   BiNode head = convertToCircular(root);
39   head.node1.node2 = null;
40   head.node1 = null;
41   return head;
42 }
```

Стоит заметить, что основные части кода были перемещены в `convertToCircular`. Метод `convert` вызывает этот метод для получения начала циклического связного списка, а затем разрывает цикл.

Такое решение занимает $O(N)$ времени, так как каждый узел обрабатывается в среднем один раз (или, точнее, $O(1)$ раз).

17.13. О нет! Вы только что закончили длинный документ, но из-за неудачной операции поиска/замены из документа пропали все пробелы, знаки препинания и прописные буквы. Предложение `I reset the computer. It still didn't boot!` превратилось в `iresetthecomputeritstilldidntboot`. Со знаками препинания и прописными буквами разберемся позднее, сначала нужно разделить строку на слова. Большинство слов находится в словаре, но есть и исключения. Для заданного словаря (списка строк) и документа (строки) разработайте алгоритм, выполняющий оптимальную деконкатенацию строки. Алгоритм должен минимизировать число нераспознанных последовательностей символов.

Пример:

Ввод: `"jesslookedjustliketimherbrother"`

Вывод: `"JESS looked just like TIM her brother"` (7 нераспознанных символов)

РЕШЕНИЕ

Одни интервьюеры предпочитают сразу переходить к сути дела и задают конкретные задачи. Другие предоставляют много лишней информации, как в нашем случае. Поэтому давайте разберемся, о чем идет речь.

Фактически требуется разбить строку на слова так, чтобы в ходе разбора было задействовано максимальное количество слов.

Обратите внимание, что мы не пытаемся «понять» строку. В результате разбора строки `thisisawesome` результат `this is a we some` ничем не отличается от `this is awesome`.

Метод «грубой силы»

Ключ к этой задаче — поиск определения решения (то есть разобранной строки) в контексте подзадач. Один из возможных способов основан на рекурсии по строке.

Прежде всего нужно решить, где следует вставить первый пробел. После первого символа? После второго? Третьего?

Представьте строку вида `thisismikesfavoritefood`. В какую позицию следует вставить первый пробел?

- Если вставить пробел после `t`, получаем один недействительный символ.
- После `th` — два недействительных символа.
- После `thi` — три недействительных символа.
- После `this` получаем полное слово. Количество недействительных символов равно 0.

- После `thisi` — пять недействительных символов.

И так далее.

После выбора позиции первого пробела мы можем рекурсивно выбрать позицию второго пробела, затем третьего и т. д., пока обработка строки не будет завершена. Мы выбираем лучший вариант (с наименьшим количеством недействительных символов) и возвращаем его.

Что должна возвращать функция? Нам понадобится как количество недействительных символов на пути рекурсии, так и фактические данные разбора. Чтобы вернуть оба значения, можно воспользоваться специальным классом `ParseResult`.

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult r = split(dictionary, sentence, 0);
3    return r == null ? null : r.parsed;
4  }
5
6  ParseResult split(HashSet<String> dictionary, String sentence, int start) {
7    if (start >= sentence.length()) {
8      return new ParseResult(0, "");
9    }
10
11    int bestInvalid = Integer.MAX_VALUE;
12    String bestParsing = null;
13    String partial = "";
14    int index = start;
15    while (index < sentence.length()) {
16      char c = sentence.charAt(index);
17      partial += c;
18      int invalid = dictionary.contains(partial) ? 0 : partial.length();
19      if (invalid < bestInvalid) { // Оптимизация
20        /* Вставить пробел и выполнить рекурсию. Если вариант лучше
21         * текущего, заменить текущий лучший вариант. */
22        ParseResult result = split(dictionary, sentence, index + 1);
23        if (invalid + result.invalid < bestInvalid) {
24          bestInvalid = invalid + result.invalid;
25          bestParsing = partial + " " + result.parsed;
26          if (bestInvalid == 0) break; // Оптимизация
27      }
28     }
29
30      index++;
31    }
32    return new ParseResult(bestInvalid, bestParsing);
33  }
34
35  public class ParseResult {
36    public int invalid = Integer.MAX_VALUE;
37    public String parsed = " ";
38    public ParseResult(int inv, String p) {
39      invalid = inv;
40      parsed = p;
41    }
42  }
```

Здесь применяются две оптимизации:

- Строка 22: если текущее количество недействительных символов превышает лучшее из известных, мы знаем, что путь рекурсии не идеален. Исследовать его нет смысла.

- Строка 30: если имеется путь с нулем недействительных символов, улучшить его уже не удастся. Можно спокойно выбрать этот путь.

За какое время выполняется этот алгоритм? Дать практическую оценку достаточно трудно, так как она зависит от (английского) языка.

Попробуйте представить странный язык, в котором отслеживаются практически все пути рекурсии. В этом случае на каждом символе принимаются оба решения. При `n` символов время выполнения составит $O(2^n)$.

Оптимизация

Как правило, экспоненциальное время выполнения для рекурсивного алгоритма можно оптимизировать посредством мемоизации (то есть кэширования результатов). Для этого необходимо выделить общие подзадачи.

Где перекрываются пути рекурсии? Иначе говоря, где находятся общие подзадачи?

Вспомните строку `thisismikesfavoritefood`. Снова будем считать, что любая комбинация является действительным словом.

В этом случае мы пытаемся вставить первый пробел после `t`, а также после `th` (и многих других букв). Подумайте, каким будет следующий выбор.

```
split(thisismikesfavoritefood) ->
      t  + split(hisismikesfavoritefood)
  ИЛИ th + split(isismikesfavoritefood)
  ИЛИ ...
split(hisismikesfavoritefood) ->
     h + split(isismikesfavoritefood)
    ИЛИ ...
...
```

Добавление пробела после `t` и `h` ведет к тому же пути рекурсии, что и вставка пробела после `th`. Нет смысла вычислять `split(isismikesfavoritefood)` дважды, если это приведет к одному результату.

Вместо этого результат следует кэшировать. Для этого можно воспользоваться хеш-таблицей, связывающей текущую подстроку с объектом `ParseResult`.

Вообще говоря, использовать в качестве ключа текущую подстроку не обязательно. Начальный индекс строки предоставляет достаточно информации — ведь если бы мы собирались использовать подстроку, то в действительности использовали бы `sentence.substring(start, sentence.length)`. Хеш-таблица будет связывать начальный индекс с лучшим разбором от этого индекса до конца строки.

А раз ключом является начальный индекс, полноценная хеш-таблица вообще не нужна. Можно просто воспользоваться массивом объектов `ParseResult`; это также решит задачу установления соответствия между индексом и объектом.

Код фактически идентичен функции, приведенной ранее, но теперь в нем используется таблица `memo` (кэш).

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult[] memo = new ParseResult[sentence.length()];
3    ParseResult r = split(dictionary, sentence, 0, memo);
4    return r == null ? null : r.parsed;
5  }
6
```

```
ParseResult split(HashSet<String> dictionary, String sentence, int start,
                  ParseResult[] memo) {
  if (start >= sentence.length()) {
    return new ParseResult(0, "");
  } if (memo[start] != null) {
    return memo[start];
  }

  int bestInvalid = Integer.MAX_VALUE;
  String bestParsing = null;
  String partial = "";
  int index = start;
  while (index < sentence.length()) {
    char c = sentence.charAt(index);
    partial += c;
    int invalid = dictionary.contains(partial) ? 0 : partial.length();
    if (invalid < bestInvalid) { // Оптимизация
      /* Вставить пробел и выполнить рекурсию. Если вариант лучше
       * текущего, заменить текущий лучший вариант. */
      ParseResult result = split(dictionary, sentence, index + 1, memo);
      if (invalid + result.invalid < bestInvalid) {
        bestInvalid = invalid + result.invalid;
        bestParsing = partial + " " + result.parsed;
        if (bestInvalid == 0) break; // Оптимизация
      }
     }

     index++;
  }
  memo[start] = new ParseResult(bestInvalid, bestParsing);
  return memo[start];
}
```

Найти оценку времени выполнения этого решения еще сложнее, чем для предыдущего. И снова рассмотрим аномальный случай, в котором любая комбинация выглядит как действительное слово.

Для получения нужной оценки можно понять, что `split(i)` будет вычисляться только один раз для каждого значения `i`. Что произойдет при вызове `split(i)`, если предположить, что мы уже выполнили вызовы от `split(i+1)` до `split(n-1)`?

```
split(i) -> calls:
  split(i + 1)
  split(i + 2)
  split(i + 3)
  split(i + 4)
  ...
  split(n - 1)
```

Все рекурсивные вызовы уже были вычислены, поэтому они просто немедленно возвращают управление. Выполнение `n - i` вызовов, каждый из которых занимает время $O(1)$, потребует времени $O(n - i)$. Это означает, что `split(i)` потребует времени не более $O(i)$.

Аналогичную логику можно применить к `split(i - 1)`, `split(i - 2)` и т. д. Если мы делаем 1 вызов для вычисления `split(n - 1)`, 2 вызова для вычисления `split(n - 2)`,

3 вызова для вычисления `split(n - 3)`, ..., `n` вызовов для вычисления `split(0)`, то сколько всего вызовов будет сделано? Фактически речь идет о сумме чисел от 1 до `n`, которая равна $O(n^2)$. Следовательно, время выполнения этой функции составит $O(n^2)$.

17.14. Разработайте алгоритм для поиска наименьших K чисел в массиве.

РЕШЕНИЕ

К решению этой задачи можно подходить несколькими разными способами. Мы рассмотрим три варианта: сортировку, max-кучу и алгоритм ранжированного выбора.

Некоторые из этих алгоритмов требуют модификации массива. Обсудите этот момент с интервьюером. Впрочем, даже если модификация исходного массива недопустима, ничто не мешает создать копию и модифицировать ее — это не отразится на общей сложности алгоритма.

Решение 1. Сортировка

Мы можем отсортировать элементы по возрастанию, а затем взять первые `k` элементов.

```
1  int[] smallestK(int[] array, int k) {
2    if (k <= 0 || k > array.length) {
3      throw new IllegalArgumentException();
4    }
5
6    /* Сортировка массива. */
7    Arrays.sort(array);
8
9    /* Копирование первых k элементов. */
10    int[] smallest = new int[k];
11    for (int i = 0; i < k; i++) {
12      smallest[i] = array[i];
13    }
14    return smallest;
15  }
```

Временная сложность равна $O(n \log(n))$.

Решение 2. Max-куча

Также для решения задачи можно воспользоваться max-кучей. Сначала создается max-куча (с наибольшим элементом на вершине) для первого миллиона чисел.

Затем начинается перебор списка. Каждый элемент, если он меньше корня, вставляется в кучу, после чего удаляется наибольший элемент (которым будет корень).

В конце обхода будет получена куча, содержащая миллион наименьших чисел. Этот алгоритм выполняется за время $O(n \log(m))$, где m — количество искомых значений.

```
1  int[] smallestK(int[] array, int k) {
2    if (k <= 0 || k > array.length) {
3      throw new IllegalArgumentException();
4    }
5
6    PriorityQueue<Integer> heap = getKMaxHeap(array, k);
```

```
7    return heapToIntArray(heap);
8  }
9
10 /* Создание кучи из k наименьших элементов. */
11 PriorityQueue<Integer> getKMaxHeap(int[] array, int k) {
12   PriorityQueue<Integer> heap =
13     new PriorityQueue<Integer>(k, new MaxHeapComparator());
14   for (int a : array) {
15     if (heap.size() < k) { // Если осталось место
16       heap.add(a);
17     } else if (a < heap.peek()) { // Куча заполнена, элемент больше
18       heap.poll(); // Удалить корень
19       heap.add(a); // Вставить новый элемент
20    }
21   }
22   return heap;
23 }
24
25 /* Преобразование кучи в целочисленный массив. */
26 int[] heapToIntArray(PriorityQueue<Integer> heap) {
27   int[] array = new int[heap.size()];
28   while (!heap.isEmpty()) {
29     array[heap.size() - 1] = heap.poll();
30   }
31   return array;
32 }
33
34 class MaxHeapComparator implements Comparator<Integer> {
35     public int compare(Integer x, Integer y) {
36         return y - x;
37     }
38 }
```

Для реализации функциональности кучи в Java используется класс `PriorityQueue`. По умолчанию он работает как min-куча, в которой на вершине находится наименьший элемент. Чтобы на вершине находился наибольший элемент, достаточно передать другой компаратор.

Решение 3. Алгоритм ранжированного выбора (для уникальных элементов)

Алгоритм ранжированного выбора часто применяется для нахождения `i`-го наименьшего (или наибольшего) элемента массива за линейное время.

Если элементы уникальны, то `i`-й наименьший элемент можно найти за время $O(n)$. Базовый алгоритм выглядит так:

1. Выбрать случайный элемент массива и использовать его как «пороговое значение». Произвести перегруппировку элементов вокруг порогового элемента и подсчитать количество элементов в левой части разбиения.
2. Если слева находятся ровно `i` элементов, просто вернуть наибольший элемент из левой части.
3. Если количество элементов в левой части больше `i`, повторить алгоритм для левой части массива.
4. Если количество элементов в левой части меньше `i`, то искать элемент с рангом `i-leftSize` (где `leftSize` — размер левой части).

Когда i-й наименьший элемент будет найден, все меньшие элементы находятся слева от него (так как элементы массива были перегруппированы соответствующим образом). Остается вернуть первые i элементов.

Ниже приведена реализация этого алгоритма.

```
1  int[] smallestK(int[] array, int k) {
2    if (k <= 0 || k > array.length) {
3      throw new IllegalArgumentException();
4    }
5
6    int threshold = rank(array, k - 1);
7    int[] smallest = new int[k];
8    int count = 0;
9    for (int a : array) {
10     if (a <= threshold) {
11       smallest[count] = a;
12       count++;
13     }
14   }
15   return smallest;
16 }
17
18 /* Получение элемента с заданным рангом. */
19 int rank(int[] array, int rank) {
20   return rank(array, 0, array.length - 1, rank);
21 }
22
23 /* Получение элемента с рангом между левым и правым индексами. */
24 int rank(int[] array, int left, int right, int rank) {
25   int pivot = array[randomIntInRange(left, right)];
26   int leftEnd = partition(array, left, right, pivot);
27   int leftSize = leftEnd - left + 1;
28   if (rank == leftSize - 1) {
29     return max(array, left, leftEnd);
30   } else if (rank < leftSize) {
31     return rank(array, left, leftEnd, rank);
32   } else {
33     return rank(array, leftEnd + 1, right, rank - leftSize);
34   }
35 }
36
37 /* Перегруппировка массива вокруг значения pivot таким образом,
38  * что все элементы <= pivot предшествуют элементам > pivot. */
39 int partition(int[] array, int left, int right, int pivot) {
40   while (left <= right) {
41     if (array[left] > pivot) {
42       /* Левая половина больше опорного значения pivot. Переместить
43        * ее направо, где она должна находиться. */
44       swap(array, left, right);
45       right--;
46     } else if (array[right] <= pivot) {
47       /* Правая половина меньше опорного значения pivot. Переместить
48        * ее налево, где она должна находиться. */
49       swap(array, left, right);
50       left++;
```

```
51      } else {
52        /* Левая и правая половины на своих местах; расширить их. */
53        left++;
54        right--;
55      }
56    }
57    return left - 1;
58  }
59
60  /* Получение случайного числа в заданном диапазоне (включительно). */
61  int randomIntInRange(int min, int max) {
62    Random rand = new Random();
63    return rand.nextInt(max + 1 - min) + min;
64  }
65
66  /* Поменять местами значения с индексами i и j. */
67  void swap(int[] array, int i, int j) {
68    int t = array[i];
69    array[i] = array[j];
70    array[j] = t;
71  }
72
73  /* Получение наибольшего элемента в массиве между индексами left и right. */
74  int max(int[] array, int left, int right) {
75    int max = Integer.MIN_VALUE;
76    for (int i = left; i <= right; i++) {
77      max = Math.max(array[i], max);
78    }
79    return max;
80  }
```

Если элементы не уникальны, алгоритм придется слегка доработать.

Решение 4. Алгоритм ранжированного выбора (если элементы не уникальны)

Главное изменение связано с функцией перегруппировки. Теперь при перегруппировке массива вокруг опорного элемента его содержимое разбивается на три блока: меньше опорного значения, равные ему и большие.

Для этого также придется внести изменения в процедуру ранжирования: теперь мы сравниваем размеры левой и средней части.

```
1  class PartitionResult {
2    int leftSize, middleSize;
3    public PartitionResult(int left, int middle) {
4      this.leftSize = left;
5      this.middleSize = middle;
6    }
7  }
8
9  int[] smallestK(int[] array, int k) {
10   if (k <= 0 || k > array.length) {
11     throw new IllegalArgumentException();
12   }
13
14   /* Получение элемента с рангом k - 1. */
15   int threshold = rank(array, k - 1);
```

```
16
17    /* Копирование элементов, меньших опорного значения. */
18    int[] smallest = new int[k];
19    int count = 0;
20    for (int a : array) {
21      if (a < threshold) {
22        smallest[count] = a;
23        count++;
24      }
25    }
26
27    /* Если еще осталось место, оно предназначено для элементов,
28     * равных пороговому. Скопировать их. */
29    while (count < k) {
30      smallest[count] = threshold;
31      count++;
32    }
33
34    return smallest;
35  }
36
37  /* Поиск значения с рангом k в массиве. */
38  int rank(int[] array, int k) {
39    if (k >= array.length) {
40      throw new IllegalArgumentException();
41    }
42    return rank(array, k, 0, array.length - 1);
43  }
44
45  /* Поиск значения с рангом k в подмассиве от start до end. */
46  int rank(int[] array, int k, int start, int end) {
47    /* Перегруппировка массива вокруг произвольного порогового значения. */
48    int pivot = array[randomIntInRange(start, end)];
49    PartitionResult partition = partition(array, start, end, pivot);
50    int leftSize = partition.leftSize;
51    int middleSize = partition.middleSize;
52
53    /* Поиск в части массива. */
54    if (k < leftSize) { // Ранг k в левой половине
55      return rank(array, k, start, start + leftSize - 1);
56    } else if (k < leftSize + middleSize) { // Ранг k в середине
57      return pivot; // Середина состоит из пороговых значений
58    } else { // Ранг k в правой половине
59      return rank(array, k - leftSize - middleSize, start + leftSize + middleSize,
60                  end);
61    }
62  }
63
64  /* Перегруппировка результата на < pivot, = pivot -> большие pivot. */
65  PartitionResult partition(int[] array, int start, int end, int pivot) {
66    int left = start; /* Остается у (правого) края левой части. */
67    int right = end;  /* Остается у (левого) края правой стороны. */
68    int middle = start; /* Остается у (правого) края средней части. */
69    while (middle <= right) {
70      if (array[middle] < pivot) {
71        /* Середина меньше pivot. Левая часть меньше либо равна
72         * pivot. В любом случае поменять их местами, после чего
```

```
73          * переместить middle и left на 1. */
74         swap(array, middle, left);
75         middle++;
76         left++;
77       } else if (array[middle] > pivot) {
78         /* Середина больше pivot. Правая часть может содержать любое
79          *  значение. Поменять их местами. Теперь мы знаем, что новая
80          * правая часть больше pivot. Переместить right на 1.*/
81         swap(array, middle, right);
82         right--;
83       } else if (array[middle] == pivot) {
84         /* Середина равна pivot. Переместить на 1. */
85         middle++;
86       }
87     }
88
89     /* Вернуть размеры left и middle. */
90     return new PartitionResult(left - start, right - left + 1);
91   }
```

Обратите внимание на то, что изменения также вносятся в `smallestK`. Мы не можем просто скопировать все элементы, меньшие либо равные порогу, в массив. Так как в массиве могут присутствовать дубликаты, в нем могут найтись более k элементов, меньших либо равных пороговому. Проблема решается достаточно просто: сначала копируются меньшие элементы, а затем массив заполняется с размещением равных элементов в конце.

17.15. Для заданного списка слов напишите алгоритм поиска самого длинного слова, образованного другими словами, входящими в список.

Пример:

Ввод: cat, banana, dog, nana, walk, walker, dogwalker

Вывод: dogwalker

РЕШЕНИЕ

Задача кажется сложной, поэтому давайте ее упростим. Допустим, нужно найти самое длинное слово, составленное из двух других слов списка.

Такую задачу можно решить перебором списка от самого длинного до самого короткого слова. Каждое слово можно разбить на возможные пары слов и проверить, содержатся ли обе части (левая и правая) в списке.

Псевдокод выглядит примерно так:

```
1  String getLongestWord(String[] list) {
2    String[] array = list.SortByLength();
3    /* Создание коллекции HashMap для удобства поиска */
4    HashMap<String, Boolean> map = new HashMap<String, Boolean>;
5
6    for (String str : array) {
7      map.put(str, true);
8    }
9
```

```
    for (String s : array) {
      // Разделить на все возможные пары
      for (int i = 1; i < s.length(); i++) {
        String left = s.substring(0, i);
        String right = s.substring(i);
        // Убедиться в том, что обе стороны присутствуют в массиве
        if (map[left] == true && map[right] == true) {
          return s;
      }
     }
    }
    return str;
  }
```

Этот код отлично работает, если нужно найти слово, составленное из двух других слов. Но что если количество слов будет произвольным?

В этом случае можно слегка модифицировать алгоритм: вместо того, чтобы искать правую часть в массиве, можно рекурсивно проверить, не является ли правая часть составной.

Следующий код реализует этот алгоритм:

```
String printLongestWord(String arr[]) {
  HashMap<String, Boolean> map = new HashMap<String, Boolean>();
  for (String str : arr) {
    map.put(str, true);
  }
  Arrays.sort(arr, new LengthComparator()); // Сортировка по длине
  for (String s : arr) {
    if (canBuildWord(s, true, map)) {
      System.out.println(s);
       return s;
    }
  }
  return "";
}

boolean canBuildWord(String str, boolean isOriginalWord,
                     HashMap<String, Boolean> map) {
  if (map.containsKey(str) && !isOriginalWord) {
    return map.get(str);
  }
  for (int i = 1; i < str.length(); i++) {
    String left = str.substring(0, i);
    String right = str.substring(i);
    if (map.containsKey(left) && map.get(left) == true &&
       canBuildWord(right, false, map)) {
      return true;
    }
  }
  map.put(str, false);
  return false;
}
```

Обратите внимание на небольшую оптимизацию, примененную в этом решении. Мы используем динамическое программирование/мемоизацию для кэширования

результатов. Таким образом, если нам понадобится многократно проверить слово `testingtester`, обработка будет выполнена только один раз.

Флаг `isOriginWord` управляет упомянутой оптимизацией. Метод `canBuildWord` вызывается для исходного слова и каждой подстроки. В первую очередь он проверяет, не содержится ли в кэше предварительно вычисленный результат. Однако с исходными словами существует одна проблема — для всех таких слов `map = true`, но мы не должны возвращать `true` (поскольку нельзя сказать, что слово построено из самого себя). Поэтому для обхода исходных слов приходится обходить эту проверку, устанавливая дополнительный флаг — `isOriginalWord`.

17.16. Популярная массажистка получает серию заявок на проведение массажа и пытается решить, какие из них следует принять. Между сеансами ей нужен 15-минутный перерыв, так что она не может принять заявки, следующие непосредственно друг за другом. Для заданной последовательности заявок (продолжительность всех сеансов кратна 15 минутам, перекрытий нет, перемещение невозможно) найдите оптимальный набор (с наибольшей суммарной продолжительностью в минутах), который может быть обслужен массажисткой. Верните суммарную продолжительность в минутах.

Пример:

Ввод: `{30, 15, 60, 75, 45, 15, 15, 45}`

Вывод: `180 минут ({30, 60, 45, 45})`.

РЕШЕНИЕ

Начнем с примера. Покажем его наглядно, чтобы вы лучше представили суть задачи. Числа обозначают продолжительность сеанса в минутах.

$r_0 = 75$	$r_1 = 105$	$r_2 = 120$	$r_3 = 75$	$r_4 = 90$	$r_5 = 135$

Также значения (включая перерыв) можно было бы разделить на 15-минутные интервалы, и получить массив `{5, 7, 8, 5, 6, 9}`. Такое представление данных будет эквивалентным, но сейчас мы будем использовать 1-минутные интервалы.

Оптимальное расписание для этой задачи состоит из элементов $\{r_0 = 75, r_2 = 120, r_5 = 135\}$ общей продолжительностью 330 минут. Мы намеренно выбрали пример, в котором оптимальная продолжительность сеансов не подразумевает строгого чередования элементов.

Также следует понимать, что стратегия первоочередного выбора с максимальной продолжительностью («жадная» стратегия) не всегда является оптимальной. Например, в серии вида `{45, 60, 45, 15}` элемент 60 не входит в оптимальный набор.

Решение 1. Рекурсивное

Первое, что приходит в голову, — рекурсивное решение. По сути имеется последовательность вариантов, принимаемых в процессе перебора списка решений: включать этот сеанс в расписание или нет? Если мы включаем сеанс `i`, то сеанс

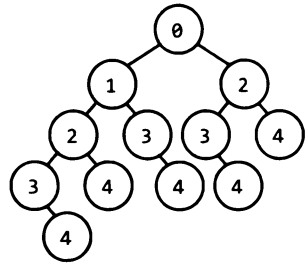
`i + 1` необходимо пропустить, так как включение двух сеансов подряд недопустимо. Включение сеанса `i + 2` возможно (хотя и не всегда является лучшим вариантом).

```
1  int maxMinutes(int[] massages) {
2    return maxMinutes(massages, 0);
3  }
4
5  int maxMinutes(int[] massages, int index) {
6    if (index >= massages.length) { // Выход за границы
7      return 0;
8    }
9
10   /* Желательно включить этот сеанс. */
11   int bestWith = massages[index] + maxMinutes(massages, index + 2);
12
13   /* Желательно не включать этот сеанс. */
14   int bestWithout = maxMinutes(massages, index + 1);
15
16   /* Return best of this subarray, starting from index. */
17   return Math.max(bestWith, bestWithout);
18 }
```

Время выполнения этого решения составляет $O(2^n)$, потому что для каждого элемента выбирается один из двух вариантов, и это повторяется n раз (где n — количество сеансов).

Пространственная сложность решения равна $O(n)$ из-за стека рекурсивных вызовов.

Также можно представить происходящее с помощью дерева рекурсивных вызовов для массива длины 5. Число в каждом узле представляет значение `index` в вызове `maxMinutes`. Например, заметим, что `maxMinutes(massages, 0)` вызывает `maxMinutes(massages, 1)` и `maxMinutes(massages, 2)`.



Как и во многих рекурсивных задачах, стоит оценить возможность применения мемоизации результатов повторяющихся подзадач. И действительно, такая возможность существует.

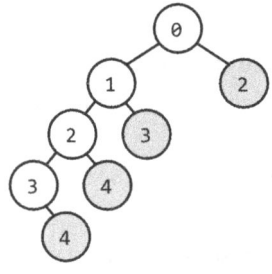
Решение 2. Рекурсия + мемоизация

Обратите внимание: `maxMinutes` многократно вызывается для одних входных данных. Например, `maxMinutes` вызывается для индекса 2, когда мы решаем, нужно ли включать в расписание сеанс 0. Такой же вызов используется, когда мы решаем, нужно ли включать в расписание сеанс 1; результат следует кэшировать.

Кэш мемоизации просто связывает индекс с результатом `maxMinutes`. Следовательно, простого массива будет достаточно.

```
1  int maxMinutes(int[] massages) {
2    int[] memo = new int[massages.length];
3    return maxMinutes(massages, 0, memo);
4  }
5
6  int maxMinutes(int[] massages, int index, int[] memo) {
7    if (index >= massages.length) {
8      return 0;
9    }
10
11    if (memo[index] == 0) {
12      int bestWith = massages[index] + maxMinutes(massages, index + 2, memo);
13      int bestWithout = maxMinutes(massages, index + 1, memo);
14      memo[index] = Math.max(bestWith, bestWithout);
15    }
16
17    return memo[index];
18  }
```

Чтобы оценить время выполнения, мы нарисуем то же дерево рекурсивных вызовов, что и прежде. Вызовы, которые немедленно возвращают управление, обозначены серым цветом, а вызовы, которые вообще никогда не происходят, полностью удалены.



Если изобразить дерево большего размера, на нем будет проявляться та же закономерность. Дерево имеет линейную структуру, с единственной длинной ветвью у левого края; таким образом, время выполнения и затраты памяти составляют $O(n)$. Затраты памяти обусловлены использованием как стека рекурсивных вызовов, так и таблицы мемоизации.

Решение 3. Итерации

Можно ли усовершенствовать этот результат? Конечно, временную сложность улучшить не удастся, потому что алгоритм должен проверить каждый сеанс. Однако пространственную сложность улучшить можно, но для этого придется обойтись без рекурсии.

Еще раз рассмотрим первый пример.

$r_0 = 30$	$r_1 = 15$	$r_2 = 60$	$r_3 = 75$	$r_4 = 45$	$r_5 = 15$	$r_6 = 15$	$r_7 = 45$

Как указано в формулировке задачи, назначать смежные сеансы нельзя.

Однако здесь нужно отметить один факт: никогда не следует пропускать три смежных сеанса. Другими словами, можно пропустить r_1 и r_2, если мы хотим назначить r_0 и r_3, однако пропускать r_1, r_2 и r_3 никогда не следует. Полученное расписание заведомо является субоптимальным, потому что его всегда можно улучшить включением среднего элемента.

Следовательно, включение r_0 приводит к гарантированному пропуску r_1 и такому же гарантированному включению либо r_2, либо r_3. Это значительно сокращает количество проверяемых вариантов и открывает путь к итеративному решению.

Проанализируем решение с рекурсией и мемоизацией и попробуем обратить логику в обратном направлении, а именно применить ее в итеративной форме. Для этого будет полезно начать с конца массива и двигаться к началу. В каждой точке ищется решение для подмассива.

- `best(7)`: как выглядит лучшее расписание для {r_7 = 45}? Только 45 минут, если взять r_7. Таким образом, `best(7)` = 45.
- `best(6)`: как выглядит лучшее расписание для {r_6 = 15, ...}? Все еще 45 минут, так что `best(6)` = 45.
- `best(5)`: как выглядит лучшее расписание для {r_5 = 15, ...}? Возможные варианты:
  - выбрать r_5 и объединить с `best(7)` = 45, либо:
  - выбрать `best(6)` = 45.

  В первом случае получаем 60 минут, `best(5)` = 60.
- `best(4)`: как выглядит лучшее расписание для {r_4 = 45, ...}? Возможные варианты:
  - выбрать r_4 = 45 и объединить с `best(6)` = 45, либо:
  - выбрать `best(5)` = 60.

  В первом случае получаем 90 минут, `best(4)` = 90.
- `best(3)`: как выглядит лучшее расписание для {r_3 = 75, ...}? Возможные варианты:
  - выбрать r_3 = 75 и объединить с `best(5)` = 60, либо:
  - выбрать `best(4)` = 90.

  В первом случае получаем 135 минут, `best(3)` = 135.
- `best(2)`: как выглядит лучшее расписание для {r_2 = 60, ...}? Возможные варианты:
  - выбрать r_2 = 60 и объединить с `best(4)` = 90, либо:
  - выбрать `best(3)` = 135.

  В первом случае получаем 150 минут, `best(2)` = 150.
- `best(1)`: как выглядит лучшее расписание для {r_1 = 15, ...}? Возможные варианты:
  - выбрать r_1 = 15 и объединить с `best(3)` = 135, либо:
  - выбрать `best(2)` = 150.

В любом случае `best(1)` = 150.

- `best(0)`: как выглядит лучшее расписание для $\{r_0 = 30, \ldots\}$? Возможные варианты:
  - выбрать $r_1 = 30$ и объединить с `best(2)` = 150, либо:
  - выбрать `best(1)` = 150.

  В первом случае получаем 180 минут, `best(0)` = 180.

Итак, возвращаемый результат — 180 минут.

Ниже приведена реализация этого алгоритма.

```
1  int maxMinutes(int[] massages) {
2    /* Выделяем в массиве два дополнительных элемента,
3     * чтобы избежать дополнительной проверки в строках 7 и 8. */
4    int[] memo = new int[massages.length + 2];
5    memo[massages.length] = 0;
6    memo[massages.length + 1] = 0;
7    for (int i = massages.length - 1; i >= 0; i--) {
8      int bestWith = massages[i] + memo[i + 2];
9      int bestWithout = memo[i + 1];
10      memo[i] = Math.max(bestWith, bestWithout);
11    }
12    return memo[0];
13 }
```

Время выполнения этого решения равно $O(n)$; пространственная сложность также равна $O(n)$.

Итеративность в некоторых отношениях удобна, но никакого реального «выигрыша» пока не видно. Рекурсивное решение обеспечивало ту же временную и пространственную сложность.

Решение 4. Итерации с оптимальным временем и затратами памяти

Анализ последнего решения показывает, что значения в таблице мемоизации используются лишь на непродолжительное время. Сместившись на несколько элементов относительно индекса, мы уже никогда не используем индекс этого элемента снова.

Собственно, для каждого заданного индекса `i` необходимо знать только оптимальное значение для `i + 1` и `i + 2`. Таким образом, можно полностью отказаться от таблицы и обойтись всего двумя целочисленными переменными.

```
1  int maxMinutes(int[] massages) {
2    int oneAway = 0;
3    int twoAway = 0;
4    for (int i = massages.length - 1; i >= 0; i--) {
5      int bestWith = massages[i] + twoAway;
6      int bestWithout = oneAway;
7      int current = Math.max(bestWith, bestWithout);
8      twoAway = oneAway;
9      oneAway = current;
10    }
11    return oneAway;
12 }
```

Мы приходим к лучшему из возможных решений: время $O(n)$, затраты памяти $O(1)$.

Почему мы перебирали массив в обратном направлении? Этот стандартный прием встречается во многих задачах. Впрочем, при желании можно было вести перебор и в прямом направлении. Одним людям проще воспринимать алгоритм так, другим наоборот. В этом случае вместо вопроса «Какое лучшее множество начинается с `a[i]`?» следует задаваться вопросом «Какое лучшее множество завершается в `a[i]`?».

17.17. Для заданной строки b и массива T строк с меньшим размером напишите метод для поиска в b всех меньших строк из T.

РЕШЕНИЕ

Начнем с примера:

```
T = {"is", "ppi", "hi", "sis", "i", "ssippi"}
b = "mississippi"
```

Обратите внимание: в этот пример включены некоторые строки (например, `"is"`), многократно встречающиеся в b.

Решение 1

«Наивное» решение вполне тривиально: мы просто ищем в большей строке каждое вхождение меньшей строки.

```
1  HashMapList<String, Integer> searchAll(String big, String[] smalls) {
2    HashMapList<String, Integer> lookup =
3      new HashMapList<String, Integer>();
4    for (String small : smalls) {
5      ArrayList<Integer> locations = search(big, small);
6      lookup.put(small, locations);
7    }
8    return lookup;
9  }
10
11 /* Поиск всех вхождений меньшей строки в большей строке. */
12 ArrayList<Integer> search(String big, String small) {
13   ArrayList<Integer> locations = new ArrayList<Integer>();
14   for (int i = 0; i < big.length() - small.length() + 1; i++) {
15     if (isSubstringAtLocation(big, small, i)) {
16       locations.add(i);
17     }
18   }
19   return locations;
20 }
21
22 /* Проверка нахождения small в строке big со смещением offset. */
23 boolean isSubstringAtLocation(String big, String small, int offset) {
24   for (int i = 0; i < small.length(); i++) {
25     if (big.charAt(offset + i) != small.charAt(i)) {
26       return false;
27     }
28   }
29   return true;
30 }
31
32 /* HashMapList<String, Integer> связывает String
33  * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Вместо того чтобы писать отдельную функцию `isAtLocation`, мы могли воспользоваться функциями `substring` и `equals`. Такое решение работает немного быстрее (хотя и не в *O*-записи), потому что оно позволяет избежать создания нескольких подстрок.

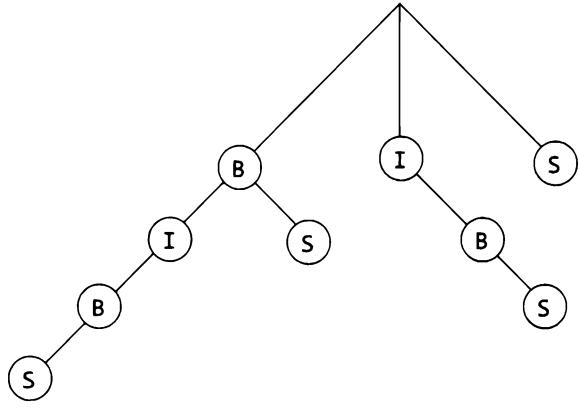
Решение выполняется за время $O(kbt)$, где `k` — длина самой длинной строки в `T`, `b` — длина большей строки, а `t` — количество меньших строк в `T`.

Решение 2

Чтобы оптимизировать это решение, следует подумать над тем, как обрабатывать все элементы `T` одновременно или каким-то образом повторно использовать готовые результаты.

Одно из возможных решений — создание структуры данных, напоминающей нагруженное дерево (trie), с использованием каждого суффикса из большей строки. Скажем, для строки `bibs` будет построен следующий список суффиксов: `bibs, ibs, bs, s`.

Ниже изображено дерево для этого списка.



Далее остается лишь провести в дереве поиск всех строк из `T`.

```
1  HashMapList<String, Integer> searchAll(String big, String[] smalls) {
2    HashMapList<String, Integer> lookup = new HashMapList<String, Integer>();
3    Trie tree = createTrieFromString(big);
4    for (String s : smalls) {
5      /* Получение завершающей позиции каждого вхождения.*/
6      ArrayList<Integer> locations = tree.search(s);
7
8      /* Поправка на начальную позицию. */
9      subtractValue(locations, s.length());
10
11      /* Вставка. */
12      lookup.put(s, locations);
13    }
14    return lookup;
15  }
16
17  Trie createTrieFromString(String s) {
18    Trie trie = new Trie();
19    for (int i = 0; i < s.length(); i++) {
```

```
20        String suffix = s.substring(i);
21        trie.insertString(suffix, i);
22      }
23      return trie;
24    }
25
26    void subtractValue(ArrayList<Integer> locations, int delta) {
27      if (locations == null) return;
28      for (int i = 0; i < locations.size(); i++) {
29        locations.set(i, locations.get(i) - delta);
30      }
31    }
32
33    public class Trie {
34      private TrieNode root = new TrieNode();
35
36      public Trie(String s) { insertString(s, 0); }
37      public Trie() {}
38
39      public ArrayList<Integer> search(String s) {
40        return root.search(s);
41      }
42
43      public void insertString(String str, int location) {
44        root.insertString(str, location);
45      }
46
47      public TrieNode getRoot() {
48        return root;
49      }
50    }
51
52    public class TrieNode {
53      private HashMap<Character, TrieNode> children;
54      private ArrayList<Integer> indexes;
55      private char value;
56
57      public TrieNode() {
58        children = new HashMap<Character, TrieNode>();
59        indexes = new ArrayList<Integer>();
60      }
61
62      public void insertString(String s, int index) {
63        indexes.add(index);
64        if (s != null && s.length() > 0) {
65          value = s.charAt(0);
66          TrieNode child = null;
67          if (children.containsKey(value)) {
68            child = children.get(value);
69          } else {
70            child = new TrieNode();
71            children.put(value, child);
72        }
73          String remainder = s.substring(1);
74          child.insertString(remainder, index + 1);
75        } else {
76          children.put('\0', null); // Завершающий символ
```

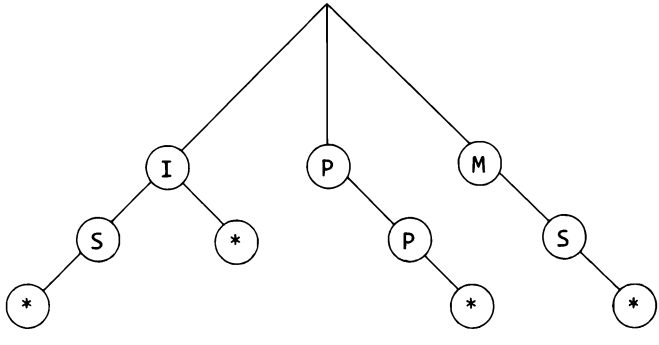
```
77     }
78    }
79
80    public ArrayList<Integer> search(String s) {
81      if (s == null || s.length() == 0) {
82        return indexes;
83      } else {
84        char first = s.charAt(0);
85        if (children.containsKey(first)) {
86          String remainder = s.substring(1);
87          return children.get(first).search(remainder);
88      }
89     }
90      return null;
91    }
92
93    public boolean terminates() {
94      return children.containsKey('\0');
95    }
96
97    public TrieNode getChild(char c) {
98      return children.get(c);
99    }
100  }
101
102  /* HashMapList<String, Integer> связывает String
103   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Создание дерева требует времени $O(b^2)$, а поиск позиций — времени $O(kt)$. Напоминаем: k — длина самой длинной строки в T, b — длина большей строки, а t — количество меньших строк в T. Таким образом, общее время выполнения составляет $O(b^2+kt)$.

Без дополнительной информации о предполагаемых входных данных прямое сравнение $O(bkt)$ (время выполнения предыдущего решения) с $O(b^2+kt)$ невозможно. При очень больших значениях b вариант $O(bkt)$ предпочтителен, а при очень большом количестве мелких строк $O(b^2+kt)$ может оказаться лучше.

Решение 3

Также возможно добавить в нагруженное дерево все меньшие строки. Например, со строками {i, is, pp, ms} дерево выглядит так, как показано ниже. Звездочка (*) означает, что предшествующий узел завершает слово.



Чтобы найти все слова, входящие в строку mississippi, мы проводим поиск по нагруженному дереву, начиная с каждого слова.

- m: поиск начинается с m, первой буквы в слове mississippi. Как только мы доберемся до mi, поиск немедленно завершается.
- Затем происходит переход к i, второму символу в mississippi. Мы видим, что i является полным словом, и добавляем его в список. Также поиск продолжается от i к is. Эта строка также является полным словом, поэтому она тоже добавляется в список. У текущего узла нет других дочерних узлов, поэтому мы переходим к следующему символу в строке mississippi.
- s: переходим к s. Узла верхнего уровня для s не существует, переходим к следующему символу.
- s: снова s. Переходим к следующему символу.
- i: еще одно вхождение i. Переходим к узлу i нагруженного дерева. Видим, что i является полным словом, и добавляем его в список. Также поиск продолжается от i к is. Эта строка также является полным словом, поэтому она тоже добавляется в список. У текущего узла нет других дочерних узлов, поэтому мы переходим к следующему символу в строке mississippi.
- s: переходим к s. Узла верхнего уровня для s не существует.
- s: снова s. Переходим к следующему символу.
- Переходим к узлу i. Видим, что i является полным словом, и добавляем его в список. Следующим символом в строке mississippi является p. Узла p в дереве не существует, обход прерывается.
- p: обнаружен символ p. Узла p в дереве нет.
- p: снова p.
- i: еще один символ i. Видим, что i является полным словом, и добавляем его в список. Больше символов в строке mississippi не осталось, обработка завершена.

Каждое обнаруженное полное «малое» слово добавляется в список вместе с позицией в большей строке (mississippi), в которой оно было обнаружено.

Ниже приведена реализация этого алгоритма.

```
1  HashMapList<String, Integer> searchAll(String big, String[] smalls) {
2    HashMapList<String, Integer> lookup = new HashMapList<String, Integer>();
3    int maxLen = big.length();
4    TrieNode root = createTreeFromStrings(smalls, maxLen).getRoot();
5
6    for (int i = 0; i < big.length(); i++) {
7      ArrayList<String> strings = findStringsAtLoc(root, big, i);
8      insertIntoHashMap(strings, lookup, i);
9    }
10
11    return lookup;
12  }
13
14  /* Вставка строк в нагруженное дерево (длина строки <= maxLen). */
15  Trie createTreeFromStrings(String[] smalls, int maxLen) {
16    Trie tree = new Trie("");
```

```
17    for (String s : smalls) {
18      if (s.length() <= maxLen) {
19        tree.insertString(s, 0);
20      }
21    }
22    return tree;
23  }
24
25  /* Поиск в нагруженном дереве строк, начинающихся с индекса start. */
26  ArrayList<String> findStringsAtLoc(TrieNode root, String big, int start) {
27    ArrayList<String> strings = new ArrayList<String>();
28    int index = start;
29    while (index < big.length()) {
30      root = root.getChild(big.charAt(index));
31      if (root == null) break;
32      if (root.terminates()) { // Полная строка, добавить в список
33        strings.add(big.substring(start, index + 1));
34      }
35      index++;
36    }
37    return strings;
38  }
39
40  /* HashMapList<String, Integer> связывает String
41   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Создание дерева требует времени $O(kt)$, а поиск всех строк — времени $O(bk)$.

Напоминаем: k — длина самой длинной строки в T, b — длина большей строки, а t – количество меньших строк в T. Таким образом, общее время выполнения составляет $O(kt+bk)$.

Решение 1 характеризовалось временем $O(kbt)$. Мы знаем, что $O(kt + bk)$ работает быстрее $O(kbt)$.

Решение 2 характеризовалось временем $O(b^2+kt)$. Так как b всегда больше k (так как в противном случае очень длинную строку k было бы невозможно найти в b), мы знаем, что решение 3 также работает быстрее решения 2.

17.18. Даны два массива, короткий (не содержащий одинаковых элементов) и длинный. Найдите в длинном массиве кратчайший подмассив, содержащий все элементы короткого массива. Элементы могут следовать в любом порядке.

Пример:

Ввод: `{1, 5, 9} | {7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}`

Вывод: `[7, 10]` (подчеркнутая часть)

Подсказки: 645, 652, 669, 681, 691, 725, 731, 741

РЕШЕНИЕ

Как обычно, решение методом «грубой силы» может стать хорошей отправной точкой. Попробуйте представить, что вам пришлось решать задачу вручную.

Как бы вы это сделали?

Воспользуемся примером из задачи. Допустим, меньшему массиву присвоено имя `smallArray`, а большему — имя `bigArray`.

Метод «грубой силы»

Медленный и «простой» способ — перебор массива `bigArray` с повторяющимися меньшими проходами по нему.

Для каждого индекса `bigArray` провести перебор в прямом направлении для нахождения каждого элемента `smallArray`. Наибольшее из следующих вхождений определит кратчайший подмассив, начинающийся с этого индекса. Назовем эту концепцию «замыканием» (closure); таким образом, замыканием является элемент, «замыкающий» полноценный подмассив, начинающийся с этого индекса. Например, замыканием индекса 3 (элемент со значением 0) является индекс 9.

Определив замыкания для каждого индекса в массиве, мы сможем найти самый короткий подмассив.

```
1  Range shortestSupersequence(int[] bigArray, int[] smallArray) {
2    int bestStart = -1;
3    int bestEnd = -1;
4    for (int i = 0; i < bigArray.length; i++) {
5      int end = findClosure(bigArray, smallArray, i);
6      if (end == -1) break;
7      if (bestStart == -1 || end - i < bestEnd - bestStart) {
8        bestStart = i;
9        bestEnd = end;
10     }
11   }
12   return new Range(bestStart, bestEnd);
13 }
14
15 /* Поиск замыкания (то есть элемента, завершающего подмассив со всеми
16  * элементами smallArray) для заданного индекса. Значение равно
17  * максимуму среди следующих позиций каждого элемента из smallArray. */
18 int findClosure(int[] bigArray, int[] smallArray, int index) {
19   int max = -1;
20   for (int i = 0; i < smallArray.length; i++) {
21     int next = findNextInstance(bigArray, smallArray[i], index);
22     if (next == -1) {
23       return -1;
24     }
25     max = Math.max(next,  max);
26   }
27   return max;
28 }
29
30 /* Поиск следующего вхождения element, начиная с индекса index. */
31 int findNextInstance(int[] array, int element, int index) {
32   for (int i = index; i < array.length; i++) {
33     if (array[i] == element) {
34       return i;
35     }
36   }
37   return -1;
38 }
39
40 public class Range {
41   private int start;
42   private int end;
43   public Range(int s, int e) {
```

```
44      start = s;
45      end = e;
46    }
47
48    public int length() { return end - start + 1; }
49    public int getStart() { return start; }
50    public int getEnd() { return end; }
51
52    public boolean shorterThan(Range other) {
53      return length() < other.length();
54    }
55  }
```

Время выполнения этого алгоритма может достигать $O(SB^2)$, где B — длина `bigString`, а S — длина `smallString`. Это объясняется тем, что для каждого из B символов выполняется работа объемом до $O(SB)$: S просмотров оставшейся части строки, которая теоретически может содержать до B символов.

Оптимизированное решение

Как можно оптимизировать это решение? Главная причина его низкой эффективности — многократные поиски. Существует ли более быстрый способ найти следующее вхождение конкретного символа для заданного индекса? Скажем, возможно ли для приведенного ниже массива быстро найти следующее вхождение 5 для каждой позиции?

`7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7`

Да, возможно. Так как мы собираемся выполнять эту операцию многократно, всю информацию можно собрать заранее за один шаг (в обратном направлении): перебрать элементы массива от конца к началу, отслеживая последнее вхождение 5.

значение	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
следующее вхождение 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x

Предварительная обработка для каждого из трех элементов `{1, 5, 9}` требует всего трех перемещений в обратном направлении.

Возможно, кто-то захочет объединить обработку в один обратный перебор, в котором рассматриваются все три значения. Такое решение кажется более быстрым, но на самом деле это не так. Выполнение операции за один шаг в обратном направлении означает три сравнения за одну итерацию. `N` перемещений по списку с тремя сравнениями не лучше `3N` перемещений с одним сравнением. Кроме того, с разделением проверок код становится более чистым.

значение	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
следующее вхождение 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
следующее вхождение 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
следующее вхождение 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x

Функция `findNextInstance` теперь может просто найти следующее вхождение по этой таблице, вместо того чтобы проводить полноценный поиск.

На самом деле решение можно даже немного упростить. По приведенной выше таблице можно быстро вычислить замыкание для каждого индекса: оно равно максимуму по каждому столбцу. Если столбец содержит `x` (обозначение отсутствия следующего вхождения), то замыкания нет.

Разность между индексом и замыканием определяет наименьший подмассив, начинающийся с этого индекса.

значение	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
следующее вхождение 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
следующее вхождение 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
следующее вхождение 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
замыкание	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x
разн.	5	4	5	6	5	4	4	3	4	3	5	4	x	x	x	x	x

Теперь остается лишь найти минимальное расстояние по этой таблице.

```
1  Range shortestSupersequence(int[] big, int[] small) {
2    int[][] nextElements = getNextElementsMulti(big, small);
3    int[] closures = getClosures(nextElements);
4    return getShortestClosure(closures);
5  }
6
7  /* Создание таблицы следующих вхождений. */
8  int[][] getNextElementsMulti(int[] big, int[] small) {
9    int[][] nextElements = new int[small.length][big.length];
10   for (int i = 0; i < small.length; i++) {
11     nextElements[i] = getNextElement(big, small[i]);
12   }
13   return nextElements;
14 }
15
16 /* Обратный перебор для построения списка следующих вхождений. */
18 int[] getNextElement(int[] bigArray, int value) {
19   int next = -1;
20   int[] nexts = new int[bigArray.length];
21   for (int i = bigArray.length - 1; i >= 0; i--) {
22     if (bigArray[i] == value) {
23       next = i;
24    }
25     nexts[i] = next;
26   }
27   return nexts;
28 }
29
30 /* Получение замыканий для всех индексов. */
31 int[] getClosures(int[][] nextElements) {
32   int[] maxNextElement = new int[nextElements[0].length];
33   for (int i = 0; i < nextElements[0].length; i++) {
34     maxNextElement[i] = getClosureForIndex(nextElements, i);
```

```
35    }
36    return maxNextElement;
37  }
38
39  /* Для заданного индекса и таблицы следующих элементов найти замыкание
40   * (которое будет равно максимуму по столбцу). */
41  int getClosureForIndex(int[][] nextElements, int index) {
42    int max = -1;
43    for (int i = 0; i < nextElements.length; i++) {
44      if (nextElements[i][index] == -1) {
45        return -1;
46      }
47      max = Math.max(max, nextElements[i][index]);
48    }
49    return max;
50  }
51
52  /* Получение кратчайшего замыкания. */
53  Range getShortestClosure(int[] closures) {
54    int bestStart = -1;
55    int bestEnd = -1;
56    for (int i = 0; i < closures.length; i++) {
57      if (closures[i] == -1) {
58        break;
59      }
60      int current = closures[i] - i;
61      if (bestStart == -1 || current < bestEnd - bestStart) {
62        bestStart = i;
63        bestEnd = closures[i];
64      }
65    }
66    return new Range(bestStart, bestEnd);
67  }
```

Время выполнения этого алгоритма может достигать $O(SB)$, где B — длина `bigString`, а S — длина `smallString`. Это объясняется тем, что при построении таблицы по массиву выполняются `S` проходов, каждый из которых занимает время $O(B)$.

Затраты памяти составляют $O(SB)$.

Усовершенствованное решение

Хотя наше решение достаточно эффективно, затраты памяти все еще можно уменьшить. Вспомните построенную таблицу:

значение	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
следующее вхождение 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
следующее вхождение 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
следующее вхождение 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
замыкание	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x

На самом деле необходимо знать лишь строку замыкания, которая определяется минимумом по всем строкам. Постоянно хранить всю информацию о каждом следующем вхождении не нужно.

Вместо этого на каждом проходе строка замыкания просто обновляется с учетом минимумов. Остальная часть алгоритма остается практически неизменной.

```
Range shortestSupersequence(int[] big, int[] small) {
  int[] closures = getClosures(big, small);
  return getShortestClosure(closures);
}

/* Получение замыканий для всех индексов. */
int[] getClosures(int[] big, int[] small) {
  int[] closure = new int[big.length];
  for (int i = 0; i < small.length; i++) {
    sweepForClosure(big, closure, small[i]);
  }
  return closure;
}

/* Обратный перебор с обновлением списка замыканий следующим
 * вхождением, если оно находится дальше текущего. */
void sweepForClosure(int[] big, int[] closures, int value) {
  int next = -1;
  for (int i = big.length - 1; i >= 0; i--) {
    if (big[i] == value) {
      next = i;
    }
    if ((next == -1 || closures[i] < next) &&
        (closures[i] != -1)) {
      closures[i] = next;
    }
  }
}

/* Получение кратчайшего замыкания. */
Range getShortestClosure(int[] closures) {
  Range shortest = new Range(0, closures[0]);
  for (int i = 1; i < closures.length; i++) {
    if (closures[i] == -1) {
      break;
    }
    Range range = new Range(i, closures[i]);
    if (!shortest.shorterThan(range)) {
      shortest = range;
    }
  }
  return shortest;
}
```

Это решение тоже выполняется за время $O(SB)$, но требует только $O(B)$ дополнительной памяти.

Альтернативное усовершенствованное решение

К решению задачи также можно подойти совершенно иначе. Допустим, существует список вхождений каждого элемента в `smallArray`.

значение	7	5	9	9	2	1	3	5	7	9	1	1	5	8	8	9	7
индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
1 -> {5, 10, 11}
5 -> {1, 7, 12}
9 -> {2, 3, 9, 15}
```

Где находится самая первая действительная подпоследовательность (содержащая 1, 5 и 9)? Чтобы узнать это, достаточно обратиться к началу каждого списка. Минимум среди начальных элементов определяет начало диапазона, а максимум — его конец. В данном случае первый диапазон имеет вид `[1, 5]`. На данный момент это «лучшая» подпоследовательность.

Как найти следующую? Следующая не будет включать индекс 1, удалим его из списка.

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {2, 3, 9, 15}
```

Следующая подпоследовательность `[2, 7]` хуже предшествующей, поэтому ее можно отбросить.

Где находится следующая подпоследовательность? Удалим минимум из предшествующей (2) и определим.

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {3, 9, 15}
```

Следующая подпоследовательность [3, 7] не лучше и не хуже текущей.

Этот процесс продолжается, пока не будут перебраны все «минимальные» подпоследовательности, начинающиеся от заданной точки.

1. Текущая подпоследовательность имеет вид [*минимум по начальным элементам, максимум по начальным элементам*]. Сравнить с лучшей подпоследовательностью и обновить при необходимости.
2. Удалить минимальный начальный элемент.
3. Повторить.

Временная сложность этого алгоритма равна $O(SB)$, так как для каждого из B элементов для поиска минимума выполняются сравнения с S другими начальными элементами.

Уже неплохо, но посмотрим, нельзя ли вычислять минимум еще быстрее.

В этих повторяющихся вызовах мы фактически берем набор элементов, находим и удаляем минимум, добавляем еще один элемент, после чего снова ищем минимум.

Для ускорения процедуры можно воспользоваться min-кучей. Сначала все начальные элементы помещаются в min-кучу, из которой удаляется минимум. Далее

проводится поиск по списку, из которого был взят этот минимум, и добавляется новый начальный элемент. Повторить.

Чтобы получить список, из которого был получен минимальный элемент, мы воспользуемся классом `HeapNode`, в котором хранится как `locationWithinList` (индекс), так и идентификатор `listId`. Таким образом при удалении минимума мы сможем вернуться к правильному списку и добавить его новый начальный элемент в кучу.

```
Range shortestSupersequence(int[] array, int[] elements) {
  ArrayList<Queue<Integer>> locations = getLocationsForElements(array, elements);
  if (locations == null) return null;
  return getShortestClosure(locations);
}

/* Получение списка очередей (связанных списков) с индексами, с которыми
 * каждый элемент из smallArray хранится в bigArray. */
ArrayList<Queue<Integer>> getLocationsForElements(int[] big, int[] small) {
  ArrayList<Queue<Integer>> allLocations = new ArrayList<Queue<Integer>>();
  for (int e : small) {
    Queue<Integer> locations = getLocations(big, e);
    if (locations.size() == 0) {
      return null;
    }
    allLocations.add(locations);
  }
  return allLocations;
}

/* Получение очереди (связного списка) индексов, под которыми элемент
 * встречается в большом массиве. */
Queue<Integer> getLocations(int[] big, int small) {
  Queue<Integer> locations = new LinkedList<Integer>();
  for (int i = 0; i < big.length; i++) {
    if (big[i] == small) {
      locations.add(i);
    }
  }
  return locations;
}

Range getShortestClosure(ArrayList<Queue<Integer>> lists) {
  PriorityQueue<HeapNode> minHeap = new PriorityQueue<HeapNode>();
  int max = Integer.MIN_VALUE;

  /* Вставка минимального элемента из каждого списка. */
  for (int i = 0; i < lists.size(); i++) {
    int head = lists.get(i).remove();
    minHeap.add(new HeapNode(head, i));
    max = Math.max(max, head);
  }

  int min = minHeap.peek().locationWithinList;
  int bestRangeMin = min;
  int bestRangeMax = max;

  while (true) {
    /* Удаление минимального узла. */
    HeapNode n = minHeap.poll();
```

```
51      Queue<Integer> list = lists.get(n.listId);
52
53      /* Сравнение диапазона с лучшим диапазоном. */
54      min = n.locationWithinList;
55      if (max - min < bestRangeMax - bestRangeMin) {
56        bestRangeMax = max;
57        bestRangeMin = min;
58      }
59
60      /* Если элементов не осталось, то подпоследовательностей
61       * не осталось, и выполнение можно прервать. */
62      if (list.size() == 0) {
63        break;
64      }
65
66      /* Добавление нового начала списка в кучу. */
67      n.locationWithinList = list.remove();
68      minHeap.add(n);
69      max = Math.max(max, n.locationWithinList);
70    }
71
72    return new Range(bestRangeMin, bestRangeMax);
73  }
```

В `getShortestClosure` перебираются B элементов, и каждый раз выполнение цикла `for` занимает время $O(\log S)$ (время вставки/удаления из кучи). Следовательно, алгоритм в худшем случае займет время $O(B \log S)$.

17.19. Имеется массив, в котором каждое из чисел от 1 до N встречается ровно один раз — кроме одного отсутствующего числа. Как найти отсутствующее число за время $O(N)$ и с затратами памяти $O(1)$? А если отсутствуют два числа?

РЕШЕНИЕ

Начнем с первой части: поиска отсутствующего числа времени $O(N)$ и с затратами памяти $O(1)$.

Часть 1. Поиск одного отсутствующего числа

В задаче установлены довольно жесткие ограничения. Мы не можем хранить все значения (это потребовало бы затрат памяти $O(N)$); тем не менее их каким-то образом необходимо «зафиксировать», чтобы мы могли выявить отсутствующее число.

Это предполагает, что со значениями должны производиться некие вычисления. Какими характеристиками они должны обладать?

- **Уникальность**. Если вычисления дают одинаковый результат для двух массивов (как указано в описании задачи), эти массивы должны быть эквивалентными (то есть содержать одинаковые пропущенные числа). Таким образом, результат вычислений должен находиться в уникальном соответствии с конкретным массивом и пропущенным числом.
- **Обратимость**. Необходим механизм перехода от результата вычислений к пропущенному числу.

- **Постоянное время**. Вычисления могут быть медленными, но они должны выполняться за постоянное время для каждого элемента массива.
- **Постоянные затраты памяти**. Вычисления могут требовать дополнительной памяти, но величина этой памяти не должна превышать $O(1)$.

Требование «уникальности» самое интересное — и самое трудное. Какие вычисления должны быть выполнены с набором чисел, чтобы отсутствующее число можно было обнаружить?

Существует несколько возможностей. Можно прибегнуть к помощи простых чисел: например, для каждого значения `x` в массиве умножить результат на `x`-е простое число. Тогда полученное значение будет заведомо уникально (так как два разных набора простых чисел не могут иметь одинакового произведения).

Обратима ли такая схема? Да. Можно взять результат и разделить его на каждое простое число: 2, 3, 5, 7 и т. д. Если для `i`-го простого числа будет получен нецелый результат, значит, значение `i` отсутствовало в массиве.

Но обеспечивает ли она постоянные затраты времени и памяти? Только если нам удастся получить `i`-е простое число за время $O(1)$ с затратами памяти $O(1)$. Нам такой способ неизвестен.

Какие еще вычисления можно проделать? Нельзя ли обойтись без простых чисел — например, просто перемножить все числа?

- **Уникальность**? Да. Представьте произведение `1*2*3*...*n`, из которого исключено одно число. Результат будет отличаться от результата, полученного при исключении любого другого числа.
- **Постоянные затраты времени и памяти**? Да.
- **Обратимость**? Давайте подумаем. Если сравнить произведение с тем, которое было бы получено без исключения числа, удастся ли найти отсутствующее число? Да, конечно. Нужно просто разделить *полное_произведение* на *фактическое_произведение*. Результат покажет, какое число отсутствует в *полном_произведении*.

Но тут возникает одна проблема: произведение получается очень, очень большим. При `n` = 20 получится где-то около 2 000 000 000 000 000 000.

Такое решение возможно, но для хранения данных придется использовать класс `BigInteger`.

```
1  int missingOne(int[] array) {
2    BigInteger fullProduct = productToN(array.length + 1);
3
4    BigInteger actualProduct = new BigInteger("1");
5    for (int i = 0; i < array.length; i++) {
6      BigInteger value = new BigInteger(array[i] + "");
7      actualProduct = actualProduct.multiply(value);
8    }
9
10    BigInteger missingNumber = fullProduct.divide(actualProduct);
11    return Integer.parseInt(missingNumber.toString());
12  }
13
14  BigInteger productToN(int n) {
15    BigInteger fullProduct = new BigInteger("1");
16    for (int i = 2; i <= n; i++) {
```

```
17      fullProduct = fullProduct.multiply(new BigInteger(i + ""));
18    }
19    return fullProduct;
20  }
```

Впрочем, во всем этом нет необходимости, так как вместо произведения можно использовать сумму. Она тоже будет уникальной.

Суммирование также обладает другим преимуществом: для вычисления суммы чисел от 1 до n уже существует выражение в закрытой форме `n(n+1)/2`.

Многие кандидаты не помнят выражения для суммы чисел от 1 до n, и это нормально. Тем не менее интервьюер может предложить вам найти ее самостоятельно. Рассуждать следует так: меньшие и большие значения в последовательности 0 + 1 + 2 + 3 + ... + n можно объединить в пары (0, n) + (1, n – 1) + (2, n – 3) и т. д. Каждая пара дает сумму n, количество таких пар равно (n + 1)/2. А если (n+1)/2 не является целым числом? В таком случае образуются n/2 пар с суммой n + 1. В любом случае получается формула n(n + 1)/2.

Переход к суммированию делает проблему переполнения намного менее вероятной, но не исключает ее полностью. Обсудите с интервьюером эту проблему и узнайте, какое решение он считает предпочтительным. Просто одного упоминания о проблеме переполнения будет достаточно для большинства интервьюеров.

Часть 2. Нахождение двух отсутствующих чисел

Эта ситуация намного сложнее. Для начала определим, что дадут предыдущие решения при отсутствии двух чисел.

- **Сумма**: сумма двух отсутствующих значений.
- **Произведение**: произведение двух отсутствующих значений.

К сожалению, знания одной суммы недостаточно. Например, если сумма равна 10, она может соответствовать парам (1, 9), (2, 8) и еще нескольким. То же самое можно сказать и о произведении.

Мы снова оказываемся в той же ситуации, в которой находились в первой части задачи: нужна формула, которая бы обеспечивала уникальность результата для всех возможных пар отсутствующих чисел.

Возможно, такая формула существует (вычисления с простыми числами работают, но не обеспечивают постоянного времени), но скорее всего, интервьюер не ожидает, что она вам известна.

Что еще можно сделать? Вернемся к суммированию и умножению. Каждый результат оставляет несколько возможностей, но оба результата сокращают этот набор до конкретных чисел.

```
x + y = сумма      -> y = сумма - x
x * y = произведение -> x(sum - x) = произведение
                       x*sum - x² = произведение
                       x*sum - x² - произведение = 0
                       -x² + x*sum - произведение = 0
```

На этой стадии можно найти x из квадратного уравнения. Зная x, можно вычислить y.

В действительности можно применить и другие вычисления. Собственно, практически любые «нелинейные» вычисления дадут значения x и y.

Так и поступим. Вместо произведения 1 * 2 * ... * n можно использовать сумму квадратов: $1^2 + 2^2 + ... + n^2$. Это сделает использование `BigInteger` менее критичным, так как код будет выполняться для меньших значений. Обсудите с интервьюером, важно это или нет.

```
x + y = s       -> y = s - x
x2 + y2 = t     -> x² + (s-x)² = t
                    2x² - 2sx + s²-t = 0
```

Вспомните формулу квадратного уравнения:

```
x = [-b +- sqrt(b² - 4ac)] / 2a
```

где в данном случае:

```
a = 2
b = -2s
c = s²-t
```

Реализация получается достаточно прямолинейной.

```
1  int[] missingTwo(int[] array) {
2    int max_value = array.length + 2;
3    int rem_square = squareSumToN(max_value, 2);
4    int rem_one = max_value * (max_value + 1) / 2;
5
6    for (int i = 0; i < array.length; i++) {
7      rem_square -= array[i] * array[i];
8      rem_one -= array[i];
9    }
10
11   return solveEquation(rem_one, rem_square);
12 }
13
14 int squareSumToN(int n, int power) {
15   int sum = 0;
16   for (int i = 1; i <= n; i++) {
17     sum += (int) Math.pow(i, power);
18   }
19   return sum;
20 }
21
22 int[] solveEquation(int r1, int r2) {
23   /* ax^2 + bx + c
24    * -->
25    * x = [-b +- sqrt(b^2 - 4ac)] / 2a
26    * В данном случае будет +, а не - */
27   int a = 2;
28   int b = -2 * r1;
29   int c = r1 * r1 - r2;
30
31   double part1 = -1 * b;
32   double part2 = Math.sqrt(b*b - 4 * a * c);
33   double part3 = 2 * a;
34
35   int solutionX = (int) ((part1 + part2) / part3);
36   int solutionY = r1 - solutionX;
37
38   int[] solution = {solutionX, solutionY};
```

```
39    return solution;
40  }
```

Возможно, вы заметили, что формула квадратного уравнения обычно дает два ответа (+/–), но в нашем коде используется только результат «+». Почему?

Существование «альтернативного» решения не означает, что одно решение «правильное», а другое «неправильное». Оно означает, что существуют ровно два значения x, для которых выполняется условие: $2x^2 - 2sx + (s^2 - t) = 0$.

Верно. Существуют. И где второе? А это y!

Если вам этот факт не кажется очевидным, вспомните, что x и y взаимозаменяемы. Если бы мы решали уравнение для y вместо x, то получилась бы идентичная формула: $2y^2 - 2sy + (s^2 - t) = 0$. Итак, y будет решением уравнения для x, а x будет решением уравнения для y.

17.20. Числа генерируются случайным образом и передаются методу. Напишите программу для вычисления медианы и ее обновления по мере поступления новых значений.

РЕШЕНИЕ

Одно из возможных решений основано на использовании двух приоритетных куч: max-куча (`maxHeap`) для значений ниже медианы и min-куча (`minHeap`) для значений выше медианы. Это позволит разделить элементы примерно поровну с двумя медианами — вершинами куч. Теперь найти медиану очень просто.

Что означает «примерно поровну»? «Примерно» означает, что при нечетном количестве чисел в одной из куч окажется лишнее число. Истинны следующие утверждения:

- если `maxHeap.size() > min.Heap.size()`, то `maxHeap.top()` будет медианой;
- если `maxHeap.size() == minHeap.size()`, то медианой будет среднее значение `maxHeap.top()` и `minHeap.top()`.

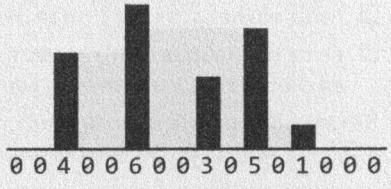
Кстати, выбранный нами способ балансировки гарантирует, что дополнительный элемент всегда будет находиться в `maxHeap`.

Алгоритм работает следующим образом. Если новое значение меньше или равно медиане, оно помещается в `maxHeap`, в противном случае оно попадает в `minHeap`. Размеры куч могут совпадать или в `maxHeap` может быть один дополнительный элемент. Это требование легко выполнить, сдвигая элемент из одной кучи в другую. Значение медианы получается за постоянное время – оно находится на вершине. Обновления занимают время $O(\log(n))$.

```
1  Comparator<Integer> maxHeapComparator, minHeapComparator;
2  PriorityQueue<Integer> maxHeap, minHeap;
3
4  void addNewNumber(int randomNumber) {
5    /* Примечание: addNewNumber обеспечивает выполнение условия
6     * maxHeap.size() >= minHeap.size() */
7    if (maxHeap.size() == minHeap.size()) {
8      if ((minHeap.peek() != null) &&
9        randomNumber > minHeap.peek()) {
10       maxHeap.offer(minHeap.poll());
```

```
        minHeap.offer(randomNumber);
      } else {
        maxHeap.offer(randomNumber);
      }
    } else {
      if(randomNumber < maxHeap.peek()) {
        minHeap.offer(maxHeap.poll());
        maxHeap.offer(randomNumber);
      }
      else {
        minHeap.offer(randomNumber);
      }
    }
  }

  double getMedian() {
    /* maxHeap всегда не меньше minHeap. Таким образом, если куча maxHeap
     * пуста, то пуста и куча minHeap. */
    if (maxHeap.isEmpty()) {
      return 0;
    }
    if (maxHeap.size() == minHeap.size()) {
      return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
    } else {
      /* Если maxHeap и minHeap имеют разные размеры, то куча maxHeap
       * содержит один лишний элемент. Вернуть вершину maxHeap.*/
      return maxHeap.peek();
    }
  }
```

17.21. Имеется гистограмма (столбцовая диаграмма). Разработайте алгоритм для вычисления объема воды, которая сможет удержаться на гистограмме (см. ниже). Предполагается, что ширина каждого столбца гистограммы равна 1.



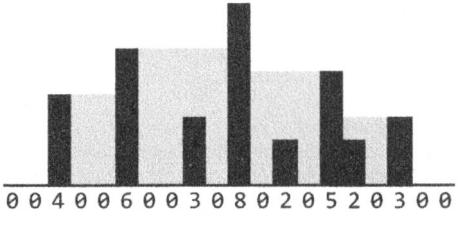
Пример (черные полосы — столбцы гистограммы, серые полосы — вода):

Ввод: {0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}

Вывод: 26

РЕШЕНИЕ

Это довольно сложная задача; создадим хороший пример, который поможет в ее решении.



Проанализируем этот пример и посмотрим, что можно узнать из него. Что именно определяет величину серых областей?

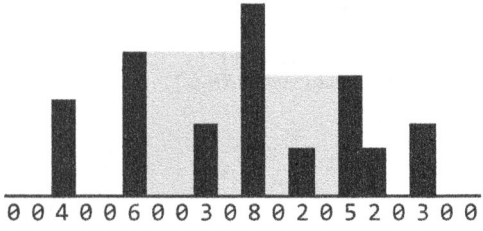
Решение 1

Возьмем самый высокий столбец, имеющий размер 8. Какую роль играет этот столбец? Может, он и самый высокий среди столбцов, но с таким же успехом он мог бы иметь высоту 100 — на объем воды это никак не повлияет.

Самый высокий столбец образует барьер для воды слева и справа от него. Однако объем воды фактически определяется следующими по высоте столбцами слева и справа.

- **Непосредственно слева от самого высокого столбца**: самый высокий столбец слева имеет высоту 6. Мы можем заполнить область между столбцами водой, но вычесть разность высот между самым высоким и следующим по высоте столбцом. Таким образом, объем воды непосредственно слева равен: (6–0) + (6–0) + (6–3) + (6–0) = 21.
- **Непосредственно справа от самого высокого столбца**: следующий по высоте столбец справа имеет высоту 5. Объем воды вычисляется по формуле: (5–0) + (5–2) + (5–0) = 13.

Но мы вычислили лишь часть общего объема.



Как насчет всего остального?

Фактически появляются две меньшие гистограммы, слева и справа. Чтобы найти объем воды в них, повторим очень похожий процесс.

1. Найти максимум. (На самом деле он уже известен — на левой подгистограмме самым высоким столбцом является правый край (6), а на правой им является левый край (5).)
2. Найти второй по высоте столбец в каждой подгистограмме. Слева это столбец 4, а справа — столбец 3.
3. Вычислить объем воды между самым высоким и вторым по высоте столбцом.
4. Повторить рекурсию.

Ниже приведена реализация этого алгоритма.

```
1  int computeHistogramVolume(int[] histogram) {
2    int start = 0;
3    int end = histogram.length - 1;
4
5    int max = findIndexOfMax(histogram, start, end);
6    int leftVolume =  subgraphVolume(histogram, start, max, true);
7    int rightVolume = subgraphVolume(histogram, max, end, false);
8
9    return leftVolume + rightVolume;
10 }
```

```
11
12  /* Вычисление объема части гистограммы. Один из максимумов равен start
13   * или end (в зависимости от isLeft). Найти второй по высоте столбец,
14   * вычислить объем между двумя столбцами, затем объем подгистограммы. */
15  int subgraphVolume(int[] histogram, int start, int end, boolean isLeft) {
16    if (start >= end) return 0;
17    int sum = 0;
18    if (isLeft) {
19      int max = findIndexOfMax(histogram, start, end - 1);
20      sum += borderedVolume(histogram, max, end);
21      sum += subgraphVolume(histogram, start, max, isLeft);
22    } else {
23      int max = findIndexOfMax(histogram, start + 1, end);
24      sum += borderedVolume(histogram, start, max);
25      sum += subgraphVolume(histogram, max, end, isLeft);
26    }
27
28    return sum;
29  }
30
31  /* Поиск самого высокого столбца между start и end. */
32  int findIndexOfMax(int[] histogram, int start, int end) {
33    int indexOfMax = start;
34    for (int i = start + 1; i <= end; i++) {
35      if (histogram[i] > histogram[indexOfMax]) {
36        indexOfMax = i;
37     }
38    }
39    return indexOfMax;
40  }
41
42  /* Вычисление объема между start и end. Предполагается, что самый
43   * высокий столбец - start, а второй по высоте - end. */
44  int borderedVolume(int[] histogram, int start, int end) {
45    if (start >= end) return 0;
46
47    int min = Math.min(histogram[start], histogram[end]);
48    int sum = 0;
49    for (int i = start + 1; i < end; i++) {
50      sum += min - histogram[i];
51    }
52    return sum;
53  }
```

Этот алгоритм занимает время $O(N^2)$ в худшем случае, где N — количество столбцов на гистограмме (так как нам приходится повторно проверять гистограмму для нахождения максимальной высоты).

Решение 2 (оптимизированное)

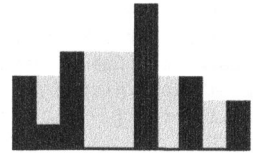
Чтобы оптимизировать рассмотренный алгоритм, подумайте, чем обусловлена его неэффективность. Главная причина — многократные вызовы `findIndexOfMax`. Это подсказывает, на чем следует сосредоточиться в процессе оптимизации.

Прежде всего следует заметить, что функции `findIndexOfMax` не передаются произвольные диапазоны — она всегда ищет максимум от одной из границ (правой или

левой). Нельзя ли быстрее узнать, чему равна максимальная высота от заданной точки до каждой границы?

Можно. Эта информация может быть получена предварительной обработкой за время $O(N)$.

За два прохода по гистограмме (справа налево и слева направо) можно построить таблицу, в которой для каждого индекса `i` хранится местонахождение максимумов справа и слева.



```
                    ИНДЕКС: 0 1 2 3 4 5 6 7 8 9
                    ВЫСОТА: 3 1 4 0 0 6 0 3 0 2
ИНДЕКС ЛЕВОГО МАКСИМУМА: 0 0 2 2 2 5 5 5 5 5
ИНДЕКС ПРАВОГО МАКСИМУМА: 5 5 5 5 5 5 7 7 9 9
```

Остальная часть алгоритма практически не изменяется.

Для хранения дополнительной информации используется класс `HistogramData`, но также можно было воспользоваться двумерным массивом.

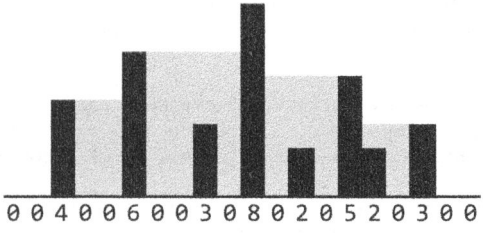
```
1  int computeHistogramVolume(int[] histogram) {
2    int start = 0;
3    int end = histogram.length - 1;
4
5    HistogramData[] data = createHistogramData(histogram);
6
7    int max = data[0].getRightMaxIndex(); // Get overall max
8    int leftVolume =  subgraphVolume(data, start, max, true);
9    int rightVolume = subgraphVolume(data, max, end, false);
10
11    return leftVolume + rightVolume;
12  }
13
14  HistogramData[] createHistogramData(int[] histo) {
15    HistogramData[] histogram = new HistogramData[histo.length];
16    for (int i = 0; i < histo.length; i++) {
17      histogram[i] = new HistogramData(histo[i]);
18    }
19
20    /* Определение индекса левого максимума. */
21    int maxIndex = 0;
22    for (int i = 0; i < histo.length; i++) {
23      if (histo[maxIndex] < histo[i]) {
24        maxIndex = i;
25     }
26      histogram[i].setLeftMaxIndex(maxIndex);
27    }
28
29    /* Определение индекса правого максимума. */
30    maxIndex = histogram.length - 1;
31    for (int i = histogram.length - 1; i >= 0; i--) {
```

```
32      if (histo[maxIndex] < histo[i]) {
33        maxIndex = i;
34      }
35      histogram[i].setRightMaxIndex(maxIndex);
36    }
37
38    return histogram;
39  }
40
41  /* Вычисление объема части гистограммы. Один из максимумов равен start
42   * или end (в зависимости от isLeft). Найти второй по высоте столбец,
43   * вычислить объем между двумя столбцами, затем объем подгистограммы. */
44  int subgraphVolume(HistogramData[] histogram, int start, int end,
45                     boolean isLeft) {
46    if (start >= end) return 0;
47    int sum = 0;
48    if (isLeft) {
49      int max = histogram[end - 1].getLeftMaxIndex();
50      sum += borderedVolume(histogram, max, end);
51      sum += subgraphVolume(histogram, start, max, isLeft);
52    } else {
53      int max = histogram[start + 1].getRightMaxIndex();
54      sum += borderedVolume(histogram, start, max);
55      sum += subgraphVolume(histogram, max, end, isLeft);
56    }
57
58    return sum;
59  }
60
61  /* Вычисление объема между start и end. Предполагается, что самый
62   * высокий столбец - start, а второй по высоте - end. */
63  int borderedVolume(HistogramData[] data, int start, int end) {
64    if (start >= end) return 0;
65
66    int min = Math.min(data[start].getHeight(), data[end].getHeight());
67    int sum = 0;
68    for (int i = start + 1; i < end; i++) {
69      sum += min - data[i].getHeight();
70    }
71    return sum;
72  }
73
74  public class HistogramData {
75    private int height;
76    private int leftMaxIndex = -1;
77    private int rightMaxIndex = -1;
78
79    public HistogramData(int v) { height = v; }
80    public int getHeight() { return height; }
81    public int getLeftMaxIndex() { return leftMaxIndex; }
82    public void setLeftMaxIndex(int idx) { leftMaxIndex = idx; };
83    public int getRightMaxIndex() { return rightMaxIndex; }
84    public void setRightMaxIndex(int idx) { rightMaxIndex = idx; };
85  }
```

Алгоритм выполняется за время $O(N)$. Так как нам приходится проверять каждый столбец, улучшить этот результат невозможно.

Решение 3 (оптимизированное и упрощенное)

Хотя временную сложность решения улучшить уже не удастся, его можно существенно упростить. Снова рассмотрим пример с учетом того, что мы уже выяснили о возможных алгоритмах.

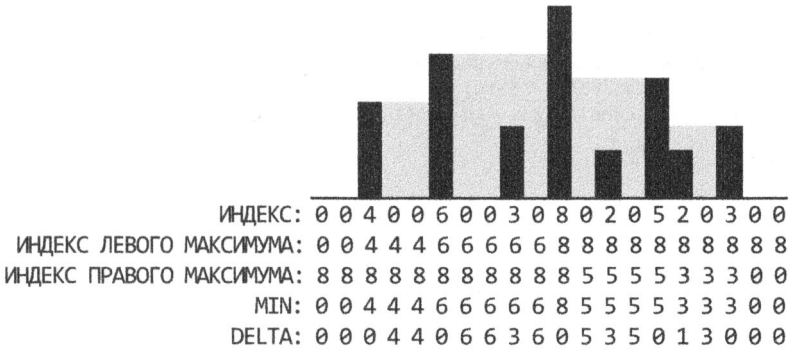


Как вы уже видели, объем воды в конкретной области определяется самыми высокими столбцами слева и справа (а конкретно более коротким слева и самым длинным справа). Например, вода заполняет область между столбцом высотой 6 и столбцом высотой 8 до высоты 6.

Таким образом, высота определяется вторым по размеру столбцом.

Общий объем воды равен сумме объемов у всех столбцов. Можно ли эффективно вычислить объем воды над каждым столбцом?

Да. В решении 2 мы заранее вычислили высоты самых высоких столбцов слева и справа для каждого индекса. Минимум этих значений определяет «уровень воды» у столбца. Разность между уровнем воды и высотой столбца будет определять объем воды.



Алгоритм состоит из нескольких простых шагов:

1. Выполнить перебор слева направо с определением максимальной обнаруженной высоты и сохранением левого максимума.
2. Выполнить перебор справа налево с определением максимальной обнаруженной высоты и сохранением правого максимума.

3. Выполнить перебор по гистограмме с вычислением минимума левого и правого максимума по каждому индексу.
4. Выполнить перебор по гистограмме с вычислением разности между каждым минимумом и высотой столбца. Просуммировать разности.

В реализации хранить столько данных не нужно, а шаги 2, 3 и 4 можно объединить в один проход. Сначала вычислим левые максимумы за один проход, а затем выполняем перебор в обратном направлении, вычисляя правый максимум. Для каждого элемента вычисляется меньшее значение по левому и правому максимуму и разность между этим значением и высотой столбца. Полученный результат прибавляется к сумме.

```
/* Перебор всех столбцов и вычисление объема воды над столбцом.
 * Объем воды над столбцом =
 *   высота - min(самый высокий столбец слева, самый высокий столбец справа)
 *
 * Вычисление левого максимума за один проход, затем повторный проход
 * для вычисления правого максимума, минимума высот и разности. */
int computeHistogramVolume(int[] histo) {
  /* Получение левого максимума */
  int[] leftMaxes = new int[histo.length];
   int leftMax = histo[0];
   for (int i = 0; i < histo.length; i++) {
     leftMax = Math.max(leftMax, histo[i]);
     leftMaxes[i] = leftMax;
   }

   int sum = 0;

   /* Получение правого максимума */
   int rightMax = histo[histo.length - 1];
   for (int i = histo.length - 1; i >= 0; i--) {
     rightMax = Math.max(rightMax, histo[i]);
     int secondTallest = Math.min(rightMax, leftMaxes[i]);

     /* Если слева и справа есть более высокие столбцы, то столбец покрыт
      * водой. Вычислить объем воды и прибавить его к сумме. */
      if (secondTallest > histo[i]) {
        sum += secondTallest - histo[i];
     }
    }

     return sum;
   }
```

Да, это весь код решения! Он также выполняется за время $O(N)$, но намного проще читается.

17.22. Для двух слов одинаковой длины, входящих в словарь, напишите метод последовательного превращения одного слова в другое с изменением одной буквы за один шаг. Каждое новое слово должно присутствовать в словаре.

Пример:
Ввод: DAMP, LIKE
Вывод: DAMP -> LAMP -> LIMP -> LIME -> LIKE

РЕШЕНИЕ

Начнем с наивного решения и постепенно перейдем к более оптимальному.

Метод «грубой силы»

Одно из возможных решений основано на преобразовании слова всеми возможными способами (конечно, с проверкой действительности слова на каждом шаге). Далее остается проверить возможность достижения конечного слова.

Таким образом, например, слово `bold` преобразуется в:

```
aold, bold, ..., zold
bald, bbld, ..., bzld
boad, bobd, ..., bozd
bola, bolb, ..., bolz
```

Если строка не является действительным словом или это слово уже посещалось, путь прерывается (то есть далее не отслеживается).

По сути речь идет о поиске в глубину, при котором «ребро» между двумя словами существует только в том случае, если они находятся на расстоянии одного изменения. Отсюда следует, что алгоритм найдет путь, но этот путь не будет самым коротким.

Если бы мы хотели найти самый короткий путь, то использовали бы поиск в ширину.

```
1  LinkedList<String> transform(String start, String stop, String[] words) {
2    HashSet<String> dict = setupDictionary(words);
3    HashSet<String> visited = new HashSet<String>();
4    return transform(visited, start, stop, dict);
5  }
6
7  HashSet<String> setupDictionary(String[] words) {
8    HashSet<String> hash = new HashSet<String>();
9    for (String word : words) {
10     hash.add(word.toLowerCase());
11   }
12   return hash;
13 }
14
15 LinkedList<String> transform(HashSet<String> visited, String startWord,
16                               String stopWord, Set<String> dictionary) {
17   if (startWord.equals(stopWord)) {
18     LinkedList<String> path = new LinkedList<String>();
19     path.add(startWord);
20     return path;
21   } else if (visited.contains(startWord) || !dictionary.contains(startWord)) {
22     return null;
23   }
24
25   visited.add(startWord);
26   ArrayList<String> words = wordsOneAway(startWord);
27
28   for (String word : words) {
29     LinkedList<String> path = transform(visited, word, stopWord, dictionary);
30     if (path != null) {
31       path.addFirst(startWord);
32       return path;
```

```
33      }
34    }
35
36    return null;
37  }
38
39  ArrayList<String> wordsOneAway(String word) {
40    ArrayList<String> words = new ArrayList<String>();
41    for (int i = 0; i < word.length(); i++) {
42      for (char c = 'a'; c <= 'z'; c++) {
43        String w = word.substring(0, i) + c + word.substring(i + 1);
44        words.add(w);
45      }
46    }
47    return words;
48  }
```

Основная неэффективность этого алгоритма связана с поиском всех строк, находящихся на расстоянии одного изменения. На данный момент мы ищем строки, находящиеся на расстоянии одного изменения, и исключаем недействительные.

В идеале хотелось бы ограничиться перебором только действительных вариантов.

Оптимизированное решение

Чтобы перебирать только действительные слова, нам понадобится способ перехода от каждого слова к списку всех связанных с ним действительных слов.

Что делает два слова «связанными» (то есть находящимися на расстоянии одного изменения)? Такие слова должны содержать одинаковые символы, кроме одного. Например, слова `ball` и `bill` находятся на расстоянии одного изменения, потому что строятся по шаблону `b_ll`. Следовательно, одно из возможных решений заключается в группировке всех слов вида `b_ll`.

Группировку можно выполнить для всего словаря; для этого шаблон, то есть слово с «универсальным символом» (вида `b_ll`), связывается со всеми словами в этой форме. Например, для очень маленького словаря вида `{all, ill, ail, ape, ale}` отображение может выглядеть так:

```
_il -> ail
_le -> ale
_ll -> all, ill
_pe -> ape
a_e -> ape, ale
a_l -> all, ail
i_l -> ill
ai_ -> ail
al_ -> all, ale
ap_ -> ape
il_ -> ill
```

Теперь, когда потребуется узнать слова, находящиеся на расстоянии одного изменения от слова вида `ale`, следует поискать по хеш-таблице соответствия для строк `_le`, `a_e` и `al_`.

В остальном алгоритм остается практически неизменным.

```
1  LinkedList<String> transform(String start, String stop, String[] words) {
2    HashMapList<String, String> wildcardToWordList =
createWildcardToWordMap(words);
3    HashSet<String> visited = new HashSet<String>();
4    return transform(visited, start, stop, wildcardToWordList);
5  }
6
7  /* Выполнить поиск в глубину от startWord к stopWord, перемещаясь
8   * от каждого слова на расстояние одного изменения. */
9  LinkedList<String> transform(HashSet<String> visited, String start, String stop,
10                               HashMapList<String, String> wildcardToWordList) {
11   if (start.equals(stop)) {
12     LinkedList<String> path = new LinkedList<String>();
13     path.add(start);
14     return path;
15   } else if (visited.contains(start)) {
16     return null;
17   }
18
19   visited.add(start);
20   ArrayList<String> words = getValidLinkedWords(start, wildcardToWordList);
21
22   for (String word : words) {
23     LinkedList<String> path = transform(visited, word, stop, wildcardToWordList);
24     if (path != null) {
25       path.addFirst(start);
26       return path;
27     }
28   }
29
30   return null;
31 }
32
33 /* Вставка в хеш-таблицу отображений вида "шаблон->слово". */
34 HashMapList<String, String> createWildcardToWordMap(String[] words) {
35   HashMapList<String, String> wildcardToWords = new HashMapList<String, String>();
36   for (String word : words) {
37     ArrayList<String> linked = getWildcardRoots(word);
38     for (String linkedWord : linked) {
39       wildcardToWords.put(linkedWord, word);
40     }
41   }
42   return wildcardToWords;
43 }
44
45 /* Получение списка шаблонов, связанных со словом. */
46 ArrayList<String> getWildcardRoots(String w) {
47   ArrayList<String> words = new ArrayList<String>();
48   for (int i = 0; i < w.length(); i++) {
49     String word = w.substring(0, i) + "_" + w.substring(i + 1);
50     words.add(word);
51   }
52   return words;
53 }
54
55 /* Получение слов, находящихся на расстоянии одного изменения. */
56 ArrayList<String> getValidLinkedWords(String word,
```

```
57      HashMapList<String, String> wildcardToWords) {
58    ArrayList<String> wildcards = getWildcardRoots(word);
59    ArrayList<String> linkedWords = new ArrayList<String>();
60    for (String wildcard : wildcards) {
61      ArrayList<String> words = wildcardToWords.get(wildcard);
62      for (String linkedWord : words) {
63        if (!linkedWord.equals(word)) {
64          linkedWords.add(linkedWord);
65      }
66     }
67    }
68    return linkedWords;
69  }
70
71  /* HashMapList<String, Integer> связывает String
72   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Такое решение работает, но его можно ускорить.

Одна из возможных оптимизаций — переключение с поиска в глубину на поиск в ширину. При нуле путей или одном пути алгоритмы работают с эквивалентной скоростью. С другой стороны, при нескольких путях поиск в ширину может работать быстрее.

Поиск в ширину находит кратчайший путь между двумя узлами, а поиск в глубину ищет любой путь. Это означает, что поиск в глубину может пройти очень долгий и извилистый путь, хотя на самом деле узлы были достаточно близки.

Оптимальное решение

Как упоминалось ранее, процедуру можно оптимизировать с помощью поиска в ширину. Будет ли она настолько быстрой, насколько это возможно? Нет.

Представьте, что путь между двумя узлами имеет длину 4. С поиском в ширину для его нахождения придется посетить около 15^4 узлов.

А что если проводить поиск от начального и конечного узла одновременно? В этом случае поиски в ширину соприкоснутся после того, как каждый из них пройдет на два уровня.

- Количество узлов, пройденных от начального узла: 15^2.
- Количество узлов, пройденных от конечного узла: 15^2.
- Всего пройдено узлов: $15^2 + 15^2$.

Такой результат намного лучше традиционного поиска в ширину. При этом нам потребуется хранить пройденный путь на каждом узле.

Чтобы реализовать этот подход, мы используем дополнительный класс `BFSData`. Он делает код чуть более понятным и позволяет применить одну структуру кода для двух одновременных поисков в ширину. В противном случае нам пришлось бы передавать набор из нескольких переменных.

```
1  LinkedList<String> transform(String startWord, String stopWord, String[] words)
{
2    HashMapList<String, String> wildcardToWordList = getWildcardToWordList(words);
3
4    BFSData sourceData = new BFSData(startWord);
```

```
  BFSData destData = new BFSData(stopWord);

  while (!sourceData.isFinished() && !destData.isFinished()) {
    /* Поиск от начального узла. */
    String collision = searchLevel(wildcardToWordList, sourceData, destData);
     if (collision != null) {
       return mergePaths(sourceData, destData, collision);
     }

     /* Поиск от конечного узла. */
     collision = searchLevel(wildcardToWordList, destData, sourceData);
     if (collision != null) {
       return mergePaths(sourceData, destData, collision);
     }
   }

   return null;
}

/* Поиск на один уровень и возвращение информации о коллизиях. */
String searchLevel(HashMapList<String, String> wildcardToWordList,
                   BFSData primary, BFSData secondary) {
   /* Поиск проводится только на один уровень. Подсчитать количество
    * узлов на уровне primary и обработать это количество узлов.
    * Узлы продолжают добавляться в конце. */
   int count = primary.toVisit.size();
   for (int i = 0; i < count; i++) {
     /* Извлечение первого узла. */
     PathNode pathNode = primary.toVisit.poll();
     String word = pathNode.getWord();

     /* Проверка посещения узла. */
     if (secondary.visited.containsKey(word)) {
       return pathNode.getWord();
     }

     /* Добавление друзей в очередь. */
     ArrayList<String> words = getValidLinkedWords(word, wildcardToWordList);
     for (String w : words) {
       if (!primary.visited.containsKey(w)) {
         PathNode next = new PathNode(w, pathNode);
         primary.visited.put(w, next);
         primary.toVisit.add(next);
       }
     }
   }
   return null;
}

LinkedList<String> mergePaths(BFSData bfs1, BFSData bfs2, String connection) {
   PathNode end1 = bfs1.visited.get(connection); // end1 -> начало
   PathNode end2 = bfs2.visited.get(connection); // end2 -> конец
   LinkedList<String> pathOne = end1.collapse(false); // в прямом направлении
   LinkedList<String> pathTwo = end2.collapse(true); // в обратном направлении
   pathTwo.removeFirst(); // Удаление связи
   pathOne.addAll(pathTwo); // Добавление второго пути
   return pathOne;
```

```
62  }
63
64  /* Методы getWildcardRoots, getWildcardToWordList и getValidLinkedWords
65   * те же, что в приведенном ранее решении. */
66
67  public class BFSData {
68    public Queue<PathNode> toVisit = new LinkedList<PathNode>();
69    public HashMap<String, PathNode> visited = new HashMap<String, PathNode>();
70
71    public BFSData(String root) {
72      PathNode sourcePath = new PathNode(root, null);
73      toVisit.add(sourcePath);
74      visited.put(root, sourcePath);
75    }
76
77    public boolean isFinished() {
78      return toVisit.isEmpty();
79    }
80  }
81
82  public class PathNode {
83    private String word = null;
84    private PathNode previousNode = null;
85    public PathNode(String word, PathNode previous) {
86      this.word = word;
87      previousNode = previous;
88    }
89
90    public String getWord() {
91      return word;
92    }
93
94    /* Обход пути и возвращение связного списка узлов. */
95    public LinkedList<String> collapse(boolean startsWithRoot) {
96      LinkedList<String> path = new LinkedList<String>();
97      PathNode node = this;
98      while (node != null) {
99        if (startsWithRoot) {
100         path.addLast(node.word);
101       } else {
102         path.addFirst(node.word);
103     }
104       node = node.previousNode;
105     }
106     return path;
107   }
108 }
109
110 /* HashMapList<String, Integer> связывает String
111  * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Оценить время выполнения этого алгоритма несколько сложнее, потому что он зависит от языка и конкретного начального и конечного слова. Один из способов выражения основан на том, что у каждого слова имеются E слов, находящихся на расстоянии одного изменения, а начальный узел находится от конечного на

расстоянии D, время выполнения составляет $O(E^{D/2})$. Такой объем работы выполняется при каждом поиске в ширину.

Конечно, объем кода получается слишком большим. Написать его на собеседовании попросту невозможно. Скорее всего, вы напишете заготовки кода `transform` и `searchLevel`, а остальное можно будет опустить.

17.23. Имеется квадратная матрица, каждая ячейка (пиксел) которой может быть черной или белой. Разработайте алгоритм поиска максимального субквадрата, у которого все четыре стороны заполнены черными пикселами.

РЕШЕНИЕ

Как и у многих задач, у этой задачи есть два решения: простое и сложное. Рассмотрим оба варианта.

«Простое» решение. $O(N^4)$

Мы знаем, что длина стороны самого большого квадрата равна `N` и существует только один квадрат размером $N \times N$. Можно проверить, существует ли такой квадрат, и вернуть управление, если он будет найден.

Если квадрат размером $N \times N$ не найден, можно попытаться найти следующий квадрат: `(N-1)×(N-1)`. Проверяя все квадраты этого размера, мы возвращаем первый найденный квадрат. Затем аналогичные операции повторяются для `N-2`, `N-3` и т. д. Так как каждый раз мы уменьшаем размер квадрата, то первый найденный квадрат будет самым большим.

Реализация выглядит так:

```
1  Subsquare findSquare(int[][] matrix) {
2    for (int i = matrix.length; i >= 1; i--) {
3      Subsquare square = findSquareWithSize(matrix, i);
4      if (square != null) return square;
5    }
6    return null;
7  }
8
9  Subsquare findSquareWithSize(int[][] matrix, int squareSize) {
10   /* На ребре длиной N существуют (N - sz + 1) квадратов длиной sz. */
11   int count = matrix.length - squareSize + 1;
12
13   /* Перебор всех квадратов с длиной стороны squareSize. */
14   for (int row = 0; row < count; row++) {
15     for (int col = 0; col < count; col++) {
16       if (isSquare(matrix, row, col, squareSize)) {
17         return new Subsquare(row, col, squareSize);
18       }
19     }
20   }
21   return null;
22 }
23
24 boolean isSquare(int[][] matrix, int row, int col, int size) {
25   // Проверка верхней и нижней границы
```

```
26    for (int j = 0; j < size; j++){
27      if (matrix[row][col+j] == 1) {
28        return false;
29      }
30      if (matrix[row+size-1][col+j] == 1){
31        return false;
32      }
33    }
34
35    // Проверка левой и правой границы
36    for (int i = 1; i < size - 1; i++){
37      if (matrix[row+i][col] == 1){
38        return false;
39      }
40      if (matrix[row+i][col+size-1] == 1) {
41        return false;
42      }
43    }
44    return true;
45  }
```

Решение с предварительной обработкой. $O(N^3)$

Низкая эффективность «простого» решения связана с тем, что мы должны произвести $O(N)$ операций при каждой проверке квадрата-кандидата. Проведя предварительную обработку, можно сократить время `isSquare` до $O(1)$; тогда алгоритм потребует $O(N^3)$ времени.

Если проанализировать работу `isSquare`, становится ясно, что функции достаточно знать, не являются ли нулевыми элементы `squareSize`, находящиеся правее (и ниже) определенных ячеек, а эту информацию можно получить заранее простым, итеративным способом.

Мы выполним проверку справа налево и снизу вверх. Для каждой ячейки нужно провести следующие вычисления:

```
если  A[r][c] является белым, A[r][c].zerosRight = 0 и A[r][c].zerosBelow = 0
иначе A[r][c].zerosRight = A[r][c + 1].zerosRight + 1
      A[r][c].zerosBelow = A[r + 1][c].zerosBelow + 1
```

Ниже приведены примеры значений для некоторой матрицы.

(нули справа, нули слева)				Исходная матрица:		
0,0	1,3	0,0		W	B	W
2,2	1,2	0,0		B	B	W
2,1	1,1	0,0		B	B	W

Теперь вместо перебора $O(N)$ элементов методу `isSquare` достаточно проверить `zerosRight` и `zerosBelow` на наличие углов.

Далее приведен код этого алгоритма. Обратите внимание, что `findSquare` и `findSquareWithSize` совпадают, если не считать вызова `processMatrix` и последующей работы с новым типом данных:

```
1  public class SquareCell {
2    public int zerosRight = 0;
3    public int zerosBelow = 0;
4    /* Объявление, get- и set-методы */
5  }
6
7  Subsquare findSquare(int[][] matrix) {
8    SquareCell[][] processed = processSquare(matrix);
9    for (int i = matrix.length; i >= 1; i--) {
10     Subsquare square = findSquareWithSize(processed, i);
11     if (square != null) return square;
12   }
13   return null;
14 }
15
16 Subsquare findSquareWithSize(SquareCell[][] processed, int size) {
17   /* Эквивалентно первому алгоритму */
18 }
19
20 boolean isSquare(SquareCell[][] matrix, int row, int col, int sz) {
21   SquareCell topLeft = matrix[row][col];
22   SquareCell topRight = matrix[row][col + sz - 1];
23   SquareCell bottomLeft = matrix[row + sz - 1][col];
24
25   /* Проверка верхней, левой, правой и нижней стороны. */
26   if (topLeft.zerosRight < sz || topLeft.zerosBelow < sz ||
27      topRight.zerosBelow < sz || bottomLeft.zerosRight < sz) {
28     return false;
29   }
30   return true;
31 }
32
33  SquareCell[][] processSquare(int[][] matrix) {
34   SquareCell[][] processed =
35     new SquareCell[matrix.length][matrix.length];
36
37   for (int r = matrix.length - 1; r >= 0; r--) {
38     for (int c = matrix.length - 1; c >= 0; c--) {
39       int rightZeros = 0;
40       int belowZeros = 0;
41       // Выполняется только для черной ячейки
42       if (matrix[r][c] == 0) {
43         rightZeros++;
44         belowZeros++;
45         // Следующий столбец в той же строке
46         if (c + 1 < matrix.length) {
47           SquareCell previous = processed[r][c + 1];
48           rightZeros += previous.zerosRight;
49       }
50         if (r + 1 < matrix.length) {
51           SquareCell previous = processed[r + 1][c];
52           belowZeros += previous.zerosBelow;
53       }
```

```
54      }
55        processed[r][c] = new SquareCell(rightZeros, belowZeros);
56      }
57    }
58    return processed;
59  }
```

17.24. Для заданной матрицы размером N×N, содержащей положительные и отрицательные числа, напишите код поиска субматрицы с максимально возможной суммой.

РЕШЕНИЕ

К решению этой задачи также можно подходить разными способами. Мы начнем с метода «грубой силы», а затем займемся оптимизацией.

Метод «грубой силы». $O(N^6)$

Как и в других задачах, связанных с поиском максимума, у этой задачи есть примитивное решение методом «грубой силы». Оно проверяет все субматрицы, вычисляет сумму каждой и ищет наибольшую.

Чтобы проверить все субматрицы и избежать дубликатов, придется пройтись по всем упорядоченным парам строк и затем по всем упорядоченным парам столбцов.

Это решение потребует времени $O(N_6)$, так как необходимо перебрать $O(N^4)$ субматриц, а проверка одной матрицы занимает $O(N^2)$ времени.

```
1  SubMatrix getMaxMatrix(int[][] matrix) {
2    int rowCount = matrix.length;
3    int columnCount = matrix[0].length;
4    SubMatrix best = null;
5    for (int row1 = 0; row1 < rowCount; row1++) {
6      for (int row2 = row1; row2 < rowCount; row2++) {
7        for (int col1 = 0; col1 < columnCount; col1++) {
8          for (int col2 = col1; col2 < columnCount; col2++) {
9            int sum = sum(matrix, row1, col1, row2, col2);
10           if (best == null || best.getSum() < sum) {
11             best = new SubMatrix(row1, col1, row2, col2, sum);
12         }
13       }
14     }
15   }
16   }
17   return best;
18 }
19
20 int sum(int[][] matrix, int row1, int col1, int row2, int col2) {
21   int sum = 0;
22   for (int r = row1; r <= row2; r++) {
23     for (int c = col1; c <= col2; c++) {
24       sum += matrix[r][c];
25   }
26   }
27   return sum;
28 }
```

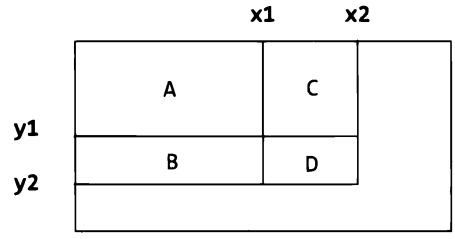
```
29
30  public class SubMatrix {
31    private int row1, row2, col1, col2, sum;
32    public SubMatrix(int r1, int c1, int r2, int c2, int sm) {
33      row1 = r1;
34      col1 = c1;
35      row2 = r2;
36      col2 = c2;
37      sum = sm;
38    }
39
40    public int getSum() {
41      return sum;
42    }
43  }
```

Выделение кода суммирования в отдельную функцию — признак хорошего стиля программирования.

Решение средствами динамического программирования. $O(N^4)$

Обратите внимание, что предыдущее решение замедляется в степени $O(N^2)$ просто потому, что суммирование элементов матрицы — очень медленная операция. Можно ли сократить это время? Да! Более того, время `computeSum` можно уменьшить до $O(1)$.

Взгляните на следующий прямоугольник:



Предположим, известны следующие значения:

```
ValD = area(point(0, 0) -> point(x2, y2))
ValC = area(point(0, 0) -> point(x2, y1))
ValB = area(point(0, 0) -> point(x1, y2))
ValA = area(point(0, 0) -> point(x1, y1))
```

Каждое значение `Val*` начинается в начале координат и заканчивается в нижнем правом углу подпрямоугольника.

Про эти значения известно следующее:

```
area(D) = ValD - area(A union C) - area(A union B) + area(A).
```

Или:

```
area(D) = ValD - ValB - ValC + ValA
```

Эти значения можно эффективно рассчитать для всех точек матрицы с использованием аналогичной логики:

```
Val(x, y) = Val(x - 1, y) + Val(y - 1, x) - Val(x - 1, y - 1)
```

Если заранее рассчитать подобные значения, затем можно эффективно найти максимальную субматрицу.

Следующий код реализует данный алгоритм.

```
1  SubMatrix getMaxMatrix(int[][] matrix) {
2    SubMatrix best = null;
3    int rowCount = matrix.length;
4    int columnCount = matrix[0].length;
5    int[][] sumThrough = precomputeSums(matrix);
6
7    for (int row1 = 0; row1 < rowCount; row1++) {
8      for (int row2 = row1; row2 < rowCount; row2++) {
9        for (int col1 = 0; col1 < columnCount; col1++) {
10         for (int col2 = col1; col2 < columnCount; col2++) {
11           int sum = sum(sumThrough, row1, col1, row2, col2);
12           if (best == null || best.getSum() < sum) {
13             best = new SubMatrix(row1, col1, row2, col2, sum);
14       }
15      }
16     }
17    }
18   }
19   return best;
20 }
21
22 int[][] precomputeSums(int[][] matrix) {
23   int[][] sumThrough = new int[matrix.length][matrix[0].length];
24   for (int r = 0; r < matrix.length; r++) {
25     for (int c = 0; c < matrix[0].length; c++) {
26       int left = c > 0 ? sumThrough[r][c - 1] : 0;
27       int top = r > 0 ? sumThrough[r - 1][c] : 0;
28       int overlap = r > 0 && c > 0 ? sumThrough[r-1][c-1] : 0;
29       sumThrough[r][c] = left + top - overlap + matrix[r][c];
30    }
31   }
32   return sumThrough;
33 }
34
35 int sum(int[][] sumThrough, int r1, int c1, int r2, int c2) {
36   int topAndLeft = r1 > 0 && c1 > 0 ? sumThrough[r1-1][c1-1] : 0;
37   int left = c1 > 0 ? sumThrough[r2][c1 - 1] : 0;
38   int top = r1 > 0 ? sumThrough[r1 - 1][c2] : 0;
39   int full = sumThrough[r2][c2];
40   return full - left - top + topAndLeft;
41 }
```

Алгоритм выполняется за время $O(N^4)$, так как он перебирает каждую пару строк и каждую пару столбцов.

Оптимизированное решение. $O(N^3)$

Как ни удивительно, существует еще более оптимальное решение. Для `R` строк и `C` столбцов задачу можно решить за время $O(R^2C)$.

Вспомните решение задачи про поиск максимального подмассива: для массива целых чисел найти подмассив с максимальной суммой. Такой максимальный подмассив можно найти за $O(N)$ времени. Используем это решение для нашей задачи.

Каждую субматрицу можно представить в виде непрерывной последовательности строк и непрерывной последовательности столбцов. Если бы нам пришлось перебирать все непрерывные последовательности строк, тогда нам было бы достаточно найти для каждой такой последовательности набор столбцов, дающих максимальную сумму:

```
1  maxSum = 0
2  foreach rowStart in rows
3    foreach rowEnd in rows
4      /* Есть много возможных субматриц с границами rowStart и rowEnd.
5       * Найти значения colStart и colEnd, обеспечивающие
6       * максимальную сумму. */
7      maxSum = max(runningMaxSum, maxSum)
8  return maxSum
```

Вопрос в том, как эффективно найти «лучшие» значения `colStart` и `colEnd`.

Представьте следующую субматрицу:

rowStart

9	-8	1	3	-2
-3	7	6	-2	4
6	-4	-4	8	-7

12 -5 3 9 -5

rowEnd

Для заданных `rowStart` и `rowEnd` нужно найти `colStart` и `colEnd`, обеспечивающие наибольшую возможную сумму. Для этого можно просуммировать каждый столбец и применить функцию `maximumSubArray`, о которой говорилось в начале задачи.

В приведенном ранее примере максимальный подмассив включает столбцы с первого по четвертый. Это означает, что максимальная субматрица проходит от (`rowStart`, первый столбец) до (`rowEnd`, четвертый столбец).

Мы приходим к псевдокоду, который выглядит примерно так:

```
1  maxSum = 0
2  foreach rowStart in rows
3    foreach rowEnd in rows
4      foreach col in columns
5        partialSum[col] = сумма от matrix[rowStart, col] до matrix[rowEnd, col]
6      runningMaxSum = maxSubArray(partialSum)
7      maxSum = max(runningMaxSum, maxSum)
8  return maxSum
```

Сумма в строках 5 и 6 вычисляется за время $R*C$ (из-за перебора от `rowStart` до `rowEnd`); получаем время выполнения $O(R^2C)$. Впрочем, это еще не все.

В строках 5 и 6 фактически суммирование `a[0]...a[i]` выполняется заново, хотя в предыдущей итерации цикла мы уже суммировали `a[0]...a[i-1]`. Избавимся от повторения выполненной работы.

```
1  maxSum = 0
2  foreach rowStart in rows
3    сброс массива partialSum
4    foreach rowEnd in rows
5      foreach col in columns
6        partialSum[col] += matrix[rowEnd, col]
7      runningMaxSum = maxSubArray(partialSum)
8    maxSum = max(runningMaxSum, maxSum)
9  return maxSum
```

Полный код выглядит так:

```
1  SubMatrix getMaxMatrix(int[][] matrix) {
2    int rowCount = matrix.length;
3    int colCount = matrix[0].length;
4    SubMatrix best = null;
5
6    for (int rowStart = 0; rowStart < rowCount; rowStart++) {
7      int[] partialSum = new int[colCount];
8
9      for (int rowEnd = rowStart; rowEnd < rowCount; rowEnd++) {
10       /* Суммирование значений в строке rowEnd. */
11       for (int i = 0; i < colCount; i++) {
12         partialSum[i] += matrix[rowEnd][i];
13     }
14
15       Range bestRange = maxSubArray(partialSum, colCount);
16       if (best == null || best.getSum() < bestRange.sum) {
17         best = new SubMatrix(rowStart, bestRange.start, rowEnd,
18                              bestRange.end, bestRange.sum);
19     }
20    }
21   }
22   return best;
23 }
24
25 Range maxSubArray(int[] array, int N) {
26   Range best = null;
27   int start = 0;
28   int sum = 0;
29
30   for (int i = 0; i < N; i++) {
31     sum += array[i];
32     if (best == null || sum > best.sum) {
33       best = new Range(start, i, sum);
34    }
35
36     /* Если running_sum < 0, продолжать бессмысленно. */
37     if (sum < 0) {
38       start = i + 1;
39       sum = 0;
40    }
41   }
42   return best;
43 }
44
45 public class Range {
```

```
46    public int start, end, sum;
47    public Range(int start, int end, int sum) {
48      this.start = start;
49      this.end = end;
50      this.sum = sum;
51    }
52  }
```

Эта задача чрезвычайно сложна. Никто не ждет, что вы полностью решите ее без помощи интервьюера.

17.25. Имеется список из миллионов слов. Разработайте алгоритм, создающий максимально возможный прямоугольник из букв, так чтобы каждая строка и каждый столбец образовывали слово (при чтении слева направо и сверху вниз). Слова могут выбираться в любом порядке, строки должны быть одинаковой длины, а столбцы — одинаковой высоты.

РЕШЕНИЕ

Многие задачи, связанные с поиском по словарю, решаются с помощью предварительной обработки. Что можно сделать в данном случае?

Если мы собираемся создать прямоугольник из слов, то длина всех строк и высота всех столбцов должны быть одинаковыми. Сгруппируем слова словаря по длине. Назовем эту группу `D`, где `D[i]` — список слов длиной `i`.

Обратите внимание, что мы ищем самый большой прямоугольник. Какой самый большой квадрат можно сформировать? Это `(length(longestWord))`^{2}.

```
1  int maxRectangle = longestWord * longestWord;
2  for z = maxRectangle to 1 {
3    для каждой пары чисел (i, j) где i*j = z {
4      /* Попытаться создать прямоугольник. return в случае успеха. */
5    }
6  }
```

Перебирая все возможные прямоугольники от самого большого до самого маленького, мы гарантируем, что первый найденный прямоугольник будет самым большим.

А теперь самая сложная часть — `makeRectangle(int l, int h)`. Этот метод пытается построить прямоугольник из слов размером `l*h`.

Можно, например, пройтись по всем упорядоченным наборам `h`-слов и затем проверить, содержат ли колонки допустимые слова. Такой метод будет работать, но очень неэффективно.

Предположим, мы пытаемся создать прямоугольник размером 6×5 и со следующими строками:

```
there
queen
pizza
.....
```

На этой стадии известно, что первый столбец начинается с `tqp`. Мы знаем, что слов, начинающихся с `tqp`, нет (по крайней мере в известных нам языках). Зачем

переходить к построению прямоугольника, если получить действительное слово все равно не удастся?

Так мы приходим к более эффективному решению. Можно построить нагруженное дерево для упрощения проверки того, является ли подстрока префиксом какого-либо слова из словаря. Затем во время построения прямоугольника мы проверяем, являются ли столбцы действительными префиксами. Если нет, то алгоритм немедленно завершает проверку, вместо того чтобы пытаться строить этот прямоугольник.

Ниже приведена реализация этого алгоритма. Код длинный и сложный, поэтому мы разберем его шаг за шагом.

Сначала выполняется предварительная обработка для группировки слов по длине. Мы создаем массив нагруженных деревьев (по одному для каждой длины слова), но построение самих деревьев откладывается до того момента, когда они понадобятся.

```
1  WordGroup[] groupList = WordGroup.createWordGroups(list);
2  int maxWordLength = groupList.length;
3  Trie trieList[] = new Trie[maxWordLength];
```

Метод `maxRectangle` — наиболее содержательная часть кода. Он начинает с самой большой возможной площади (`maxWordLength`^{2}) и пытается построить прямоугольник этого размера. В случае неудачи площадь уменьшается на 1 и повторяет попытку с новым, уменьшенным размером. Первый прямоугольник, который удастся построить, заведомо будет наибольшим из возможных.

```
1  Rectangle maxRectangle() {
2    int maxSize = maxWordLength * maxWordLength;
3    for (int z = maxSize; z > 0; z--) { // Начать с наибольшей площади
4      for (int i = 1; i <= maxWordLength; i ++ ) {
5        if (z % i == 0) {
6          int j = z / i;
7          if (j <= maxWordLength) {
8            /* Создание прямоугольника с длиной i и высотой j (i*j = z). */
9            Rectangle rectangle = makeRectangle(i, j);
10            if (rectangle != null) return rectangle;
11       }
12      }
13     }
14    }
15    return null;
16  }
```

Метод `makeRectangle`, вызываемый `maxRectangle`, пытается построить прямоугольник с заданной длиной и высотой.

```
1  Rectangle makeRectangle(int length, int height) {
2    if (groupList[length-1] == null || groupList[height-1] == null) {
3      return null;
4    }
5
6    /* Создание нагруженного дерева для длины слова */
7    if (trieList[height - 1] == null) {
8     LinkedList<String> words = groupList[height - 1].getWords();
9      trieList[height - 1] = new Trie(words);
10    }
11
```

```
12    return makePartialRectangle(length, height, new Rectangle(length));
13  }
```

Все основное начинается с метода `makePartialRectangle`. Ему передается предполагаемая итоговая длина и высота и частично сформированный прямоугольник. Если прямоугольник уже имеет итоговую длину, мы просто проверяем, образуют ли столбцы полноценные слова, и возвращаем управление.

В противном случае мы проверяем, образуют ли столбцы действительные префиксы. Если нет, то обработка немедленно прерывается, так как из этого неполного прямоугольника не удастся построить действительный прямоугольник.

Но если все идет нормально и все столбцы образуют действительные префиксы слов, мы проводим поиск по всем словам подходящей длины, присоединяем каждое к текущему прямоугольнику и рекурсивно пытаемся построить прямоугольник из текущего прямоугольника с присоединенным новым словом.

```
1  Rectangle makePartialRectangle(int l, int h, Rectangle rectangle) {
2    if (rectangle.height == h) { // Является ли полным прямоугольником?
3      if (rectangle.isComplete(l, h, groupList[h - 1])) {
4        return rectangle;
5      }
6      return null;
7    }
8
9    /* Сравнение для проверки потенциально действительного прямоугольника */
10    if (!rectangle.isPartialOK(l, trieList[h - 1])) {
11      return null;
12    }
13
14    /* Перебор всех слов подходящей длины. После добавления каждого
15     * потенциального слова пытаемся рекурсивно построить прямоугольник. */
16    for (int i = 0; i < groupList[l-1].length(); i++) {
17      /* Создание нового прямоугольника с новым словом. */
18      Rectangle orgPlus = rectangle.append(groupList[l-1].getWord(i));
19
20      /* Попытка построения прямоугольника на базе нового неполного */
21      Rectangle rect = makePartialRectangle(l, h, orgPlus);
22      if (rect != null) {
23        return rect;
24      }
25    }
26    return null;
27  }
```

Класс `Rectangle` представляет частично или полностью сформированный прямоугольник из слов. Метод `isPartialOk` проверяет, является ли прямоугольник действительным на данный момент (то есть образуют ли все столбцы префиксы слов).

Метод `isComplete` выполняет аналогичную функцию, но проверяет, образует ли каждый столбец полное слово.

```
1  public class Rectangle {
2    public int height, length;
3    public char[][] matrix;
4
5    /* Построение "пустого" прямоугольника. Длина остается постоянной,
```

```
6     * а высота изменяется с добавлением слов. */
7    public Rectangle(int l) {
8      height = 0;
9      length = l;
10   }
11
12   /* Построение прямоугольного массива букв с заданной длиной и высотой
13    * на основании заданной матрицы букв. (Предполагается, что длина
14    * и высота, переданные в аргументах, соответствуют размерам
15    * аргумента-массива.) */
16   public Rectangle(int length, int height, char[][] letters) {
17     this.height = letters.length;
18     this.length = letters[0].length;
19     matrix = letters;
20   }
21
22   public char getLetter (int i, int j) { return matrix[i][j]; }
23   public String getColumn(int i) { ...  }
24
25   /* Проверка действительности всех столбцов. Все строки заведомо
26    * действительны, так как они добавлялись из словаря. */
27   public boolean isComplete(int l, int h, WordGroup groupList) {
28     if (height == h) {
29       /* Проверяем, является ли каждый столбец словом из словаря. */
30       for (int i = 0; i < l; i++) {
31         String col = getColumn(i);
32         if (!groupList.containsWord(col)) {
33           return false;
34      }
35     }
36       return true;
37    }
38     return false;
39   }
40
41   public boolean isPartialOK(int l, Trie trie) {
42     if (height == 0) return true;
43     for (int i = 0; i < l; i++ ) {
44       String col = getColumn(i);
45       if (!trie.contains(col)) {
46         return false;
47     }
48    }
49     return true;
50   }
51
52   /* Создание нового объекта Rectangle из строк текущего прямоугольника
53    * с присоединением s. */
54   public Rectangle append(String s) { ... }
55 }
```

Класс `WordGroup` представляет собой простой контейнер для всех слов конкретной длины. Для удобства выборки слова хранятся как в хеш-таблице, так и в `ArrayList`.

Списки в `WordGroup` создаются статическим методом `createWordGroups`.

```
1  public class WordGroup {
2    private HashMap<String, Boolean> lookup = new HashMap<String, Boolean>();
3     private ArrayList<String> group = new ArrayList<String>();
```

```
4    public boolean containsWord(String s) { return lookup.containsKey(s); }
5    public int length() { return group.size(); }
6    public String getWord(int i) { return group.get(i); }
7    public ArrayList<String> getWords() { return group; }
8
9    public void addWord (String s) {
10      group.add(s);
11      lookup.put(s, true);
12    }
13
14    public static WordGroup[] createWordGroups(String[] list) {
15      WordGroup[] groupList;
16      int maxWordLength = 0;
17      /* Определение длины самого длинного слова */
18      for (int i = 0; i < list.length; i++) {
19        if (list[i].length() > maxWordLength) {
20          maxWordLength = list[i].length();
21        }
22      }
23
24      /* Группировка слов из словаря в списки слов с одинаковой длиной.
25       * В groupList[i] хранится список слов, имеющих длину (i+1). */
26      groupList = new WordGroup[maxWordLength];
27      for (int i = 0; i < list.length; i++) {
28        /* We do wordLength - 1 instead of just wordLength since this is used as
29         * an index and no words are of length 0 */
30        int wordLength = list[i].length() - 1;
31        if (groupList[wordLength] == null) {
32          groupList[wordLength] = new WordGroup();
33        }
34        groupList[wordLength].addWord(list[i]);
35      }
36      return groupList;
37    }
38  }
```

Полный код задачи, включающий код `Trie` и `TrieNode`, можно найти в архиве примеров. Задача очень сложная, и, скорее всего, вам будет достаточно написать псевдокод. Написать полный код такой задачи за короткое время практически невозможно.

17.26. Степень сходства двух документов (каждый из которых состоит из разных слов) определяется как отношение между размером пересечения и размером объединения. Например, если документы состоят из целых чисел, то степень сходства между {1, 5, 3} и {1, 7, 2, 3} равна 0,4, потому что размер пересечения равен 2, а размер объединения равен 5.

Имеется длинный список документов (с разными значениями, с каждым документом связывается идентификатор), для которых степень сходства считается «минималистической». Иначе говоря, для любых двух произвольно выбранных документов степень сходства с большой вероятностью равна 0. Разработайте алгоритм, который возвращает списки пар идентификаторов документов и степеней сходства между ними.

Выведите только те пары, у которых степень сходства больше 0. Пустые документы вообще не включаются в вывод. Для простоты можно считать, что каждый документ представлен массивом неповторяющихся целых чисел.

Пример:

Ввод:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Вывод:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

РЕШЕНИЕ

Задача довольно непростая, поэтому начнем с алгоритма методом «грубой силы». По крайней мере это поможет нам лучше понять суть задачи.

Вспомните, что каждый документ представляет собой массив «слов», каждое из которых представляет собой целое число.

Метод «грубой силы»

Алгоритм «грубой силы» сводится к простому сравнению каждого массива со всеми остальными массивами. При каждом сравнении вычисляются размеры пересечения и объединения двух массивов.

Заметим, что пара выводится только в том случае, если степень сходства больше 0. Объявление двух массивов никогда не может быть пустым (кроме случая, когда оба массива пусты, но тогда их все равно выводить не следует). Таким образом, в действительности степень сходства выводится только в том случае, если размер пересечения больше 0.

Как вычислить размер пересечения и объединения?

Под пересечением понимается количество общих элементов. Следовательно, мы можем просто перебрать первый массив (`A`) и проверить, присутствует ли каждый его элемент во втором массиве (`B`). При обнаружении элемента увеличивается переменная `intersection`.

При вычислении объединения необходимо позаботиться о том, чтобы избежать двойного подсчета элементов, входящих в оба массива. Для этого можно подсчитать все элементы `A`, не входящие в `B`, а затем добавить все элементы из `B`. В этом случае дубликаты будут учтены только один раз — в составе `B`.

Также возможно действовать иначе. При двойном подсчете элементы, входящие в пересечение (то есть присутствующие как в `A`, так и в `B`), будут включены дважды. а это означает, что задача может быть легко решена простым удалением дубликатов.

```
объединение(A, B) = A + B - пересечение(A, B)
```

Таким образом, достаточно вычислить только пересечение. Объединение (а следовательно, и степень сходства) может быть вычислено на основании этой информации.

Итак, только для сравнения двух массивов (документов) алгоритм будет выполняться за время $O(AB)$. Операция должна быть выполнена для всех пар из D документов. Если каждый документ содержит не более W слов, то время выполнения составит $O(D^2 W^2)$.

Слегка улучшенное решение методом «грубой силы»

Для получения немедленного выигрыша можно оптимизировать вычисление степени сходства двух массивов, а точнее, вычисление пересечения.

Необходимо знать количество общих элементов в двух массивах. Все элементы A заносятся в хеш-таблицу, после чего мы перебираем B и увеличиваем `intersection` при каждом обнаружении элемента из A.

Процедура выполняется за время $O(A + B)$. Если каждый массив имеет размер W, а операция выполняется для D массивов, то общие затраты времени составят $O(D^2 W)$.

Прежде чем браться за реализацию, подумаем, какие классы нам могут понадобиться. Для возвращения списка пар документов и степеней сходства будет использоваться класс `DocPair`. Возвращаемым типом будет хеш-таблица, которая связывает `DocPair` со значением `double`, представляющим степень сходства.

```
1  public class DocPair {
2    public int doc1, doc2;
3
4    public DocPair(int d1, int d2) {
5      doc1 = d1;
6      doc2 = d2;
7    }
8
9    @Override
10    public boolean equals(Object o) {
11      if (o instanceof DocPair) {
12        DocPair p = (DocPair) o;
13        return p.doc1 == doc1 && p.doc2 == doc2;
14     }
15      return false;
16    }
17
18    @Override
19    public int hashCode() { return (doc1 * 31) ^ doc2; }
20  }
```

Также будет полезно иметь класс для представления документов.

```
1  public class Document {
2    private ArrayList<Integer> words;
3    private int docId;
4 5    public Document(int id, ArrayList<Integer> w) {
6      docId = id;
7      words = w;
8    }
```

```
9
10    public ArrayList<Integer> getWords() { return words; }
11    public int getId() { return docId; }
12    public int size() { return words == null ? 0 : words.size(); }
13  }
```

Строго говоря, можно обойтись и без этого. Однако удобочитаемость кода важна, а `ArrayList<Document>` воспринимается намного проще, чем `ArrayList<ArrayList<Integer>>`.

Такой подход не только демонстрирует хороший стиль программирования, но и упрощает вашу задачу на собеседованиях, сокращая объем кода. (Вероятно, вы не станете писать определение класса `Document`, если только вас об этом не попросит интервьюер.)

```
1  HashMap<DocPair, Double> computeSimilarities(ArrayList<Document> documents) {
2    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
3    for (int i = 0; i < documents.size(); i++) {
4      for (int j = i + 1; j < documents.size(); j++) {
5        Document doc1 = documents.get(i);
6        Document doc2 = documents.get(j);
7        double sim = computeSimilarity(doc1, doc2);
8        if (sim > 0) {
9          DocPair pair = new DocPair(doc1.getId(), doc2.getId());
10          similarities.put(pair, sim);
11       }
12     }
13   }
14   return similarities;
15 }
16
17 double computeSimilarity(Document doc1, Document doc2) {
18   int intersection = 0;
19   HashSet<Integer> set1 = new HashSet<Integer>();
20   set1.addAll(doc1.getWords());
21
22   for (int word : doc2.getWords()) {
23     if (set1.contains(word)) {
24       intersection++;
25     }
26   }
27
28   double union = doc1.size() + doc2.size() - intersection;
29   return intersection / union;
30 }
```

Обратите внимание на происходящее в строке 28. Почему переменная `union` объявлена с типом `double`, хотя она явно целочисленная?

Мы сделали это для предотвращения ошибок целочисленного деления. В противном случае результат деления был бы «округлен» до целого, а степень сходства почти всегда была бы равна нулю!

Слегка улучшенное решение методом «грубой силы» (альтернативное)

Если документы были отсортированы, то пересечение двух документов можно вычислить простым перебором в порядке сортировки — по аналогии с тем, как бы вы действовали при слиянии двух отсортированных массивов.

Это потребовало бы времени $O(A + B)$. Временная сложность такая же, как у текущего алгоритма, но затраты памяти ниже. Обработка для `D` документов, каждый из которых содержит `W` слов, займет время $O(D^2 W)$.

Так как мы не знаем, были ли массивы предварительно отсортированы, можно начать с их сортировки. Это займет время $O(D * W \log W)$, а общее время выполнения составит $O(D * W \log W + D^2 W)$.

Не следует считать, что вторая часть непременно «доминирует» над первой. Все зависит от относительных значений `D` и `log W`. Следовательно, в оценку времени выполнения следует включить обе составляющие.

Оптимизированное (в определенной степени) решение

Чтобы действительно глубоко понять суть задачи, стоит создать более серьезный пример.

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10, 3}
```

Прежде всего, нельзя ли применить различные средства, позволяющие быстрее исключать потенциальные сравнения? Например, можно ли вычислить минимум и максимум для каждого массива? В этом случае мы будем знать, что массивы без перекрытий сравнивать не нужно.

Проблема в том, что это не решает проблемы времени выполнения. До настоящего момента нам удалось достичь времени выполнения $O(D^2 W)$. С таким изменением мы по-прежнему будем сравнивать все $O(D^2)$ пар, но часть $O(W)$ иногда может сокращаться до $O(1)$. При больших значениях `D` проблемы возникнут с частью $O(D^2)$.

Следовательно, сосредоточиться нужно на сокращении составляющей $O(D^2)$ — она является «узким местом» нашего решения. А конкретно это означает, что для заданного документа `docA` требуется найти все документы, обладающие некоторой степенью сходства, и это должно быть сделано без обработки каждого документа.

Что делает документ сходным с `docA`? Иначе говоря, какие характеристики определяют документы со степенью сходства >0?

Допустим, `docA` состоит из элементов `{14, 15, 100, 9, 3}`. Чтобы степень сходства другого документа была положительной, он должен содержать 14, 15, 100, 9 или 3. Как быстро получить список всех документов, содержащих хотя бы один из этих элементов?

Медленный (и на самом деле единственный) способ заключается в чтении каждого слова из каждого документа. Это потребует времени $O(DW)$, что вряд ли можно признать удовлетворительным.

Однако обратите внимание на то, что эта работа выполняется многократно. Результаты одного вызова можно повторно использовать при другом вызове.

Если построить хеш-таблицу, связывающую слово со всеми документами, содержащими это слово, можно очень быстро определить документы, пересекающиеся с `docA`.

```
1 -> 16
2 -> 19
3 -> 13, 16, 24
5 -> 16
6 -> 19
7 -> 19, 24
8 -> 19
9 -> 13, 16
...
```

Когда потребуется узнать все документы, пересекающиеся с `docA`, достаточно провести поиск всех элементов `docA` в хеш-таблице. Так будет получен список документов с некоторыми перекрытиями. Далее остается только сравнить `docA` с каждым из этих документов.

Если существуют `P` пар с неотрицательной степенью сходства и каждый документ содержит `W` слов, проверка займет время $O(PW)$ (плюс время $O(DW)$ на создание и чтение хеш-таблицы). Так как ожидается, что `P` намного меньше D^2, этот результат намного лучше предыдущего.

Оптимизированное решение (усовершенствованное)

Вернемся к предыдущему алгоритму. Можно ли повысить его эффективность?

Если взять время выполнения — $O(PW + DW)$ — скорее всего, избавиться от составляющей $O(DW)$ не удастся. К каждому слову придется обратиться хотя бы один раз, а количество слов равно $O(DW)$. Следовательно, если оптимизация возможна, она с большой вероятностью будет относиться к составляющей $O(PW)$.

Убрать часть `P` в $O(PW)$ вряд ли возможно, потому что необходимо как минимум вывести все `P` пар (что займет время $O(P)$). Итак, сосредочиться следует на части `W`. Можно ли выполнять для каждой пары сходных документов объем работы, меньший $O(W)$?

Попробуем проанализировать информацию, предоставляемую хеш-таблицей. Допустим, имеется следующий список документов:

```
12: {1, 5, 9}
13: {5, 3, 1, 8}
14: {4, 3, 2}
15: {1, 5, 9, 8}
17: {1, 6}
```

Проведя поиск элементов документа 12 в хеш-таблице, получим:

```
1 -> {12, 13, 15, 17}
5 -> {12, 13, 15}
9 -> {12, 15}
```

Полученная информация указывает на то, что документы 13, 15 и 17 имеют некоторое сходство. В текущем алгоритме теперь нужно сравнить документ 12

с документами 13, 15 и 17, чтобы узнать количество элементов, общих с документом 12 у каждого из них (то есть размер пересечения). Объединение может быть вычислено по размерам документов и пересечения, как это делалось прежде.

Однако заметим, что документ 13 встречается в хеш-таблице дважды, документ 14 — трижды, а документ 17 встречается всего один раз. Нельзя ли использовать эту информацию?

Документ 13 встречается дважды, потому что он содержит два общих элемента (1 и 5). Документ 17 встречается один раз, потому что он содержит всего один общий элемент (1). Документ 15 встречается трижды, потому что он содержит три общих элемента (1, 5 и 9). Эта информация напрямую дает размер пересечения.

Можно перебрать документы, проверить элементы по хеш-таблице и подсчитать, сколько раз каждый документ встречается в списках каждого элемента. Впрочем, существует и более прямолинейный способ.

1. Как и прежде, построить хеш-таблицу для списка документов.
2. Создать новую хеш-таблицу, которая связывает пару документов с целым числом (обозначающим размер пересечения).
3. Прочитать первую хеш-таблицу и перебрать списки документов.
4. Для каждого списка документов перебрать пары из этого списка. Увеличить счетчик пересечений для каждой пары.

Сравнить это время выполнения с предыдущим не так просто. Как вариант, можно заметить, что до этого для каждой пары с ненулевым сходством выполнялась работа $O(W)$, так как мы обращались к каждому слову в каждом документе. С этим алгоритмом мы обращаемся только к фактически пересекающимся словам. Худшие случаи остаются теми же, но для многих вариантов входных данных этот алгоритм работает быстрее.

```
1  HashMap<DocPair, Double>
2  computeSimilarities(HashMap<Integer, Document> documents) {
3    HashMapList<Integer, Integer> wordToDocs = groupWords(documents);
4    HashMap<DocPair, Double> similarities = computeIntersections(wordToDocs);
5    adjustToSimilarities(documents, similarities);
6    return similarities;
7  }
8
9  /* Создание хеш-таблицы, связывающей слова с их вхождениями. */
10 HashMapList<Integer, Integer> groupWords(HashMap<Integer, Document> documents)
{
11   HashMapList<Integer, Integer> wordToDocs = new HashMapList<Integer, Integer>();
12
13   for (Document doc : documents.values()) {
14     ArrayList<Integer> words = doc.getWords();
15     for (int word : words) {
16       wordToDocs.put(word, doc.getId());
17     }
18   }
19
20   return wordToDocs;
21 }
```

```
22
23  /* Вычисление пересечения документов: перебор каждого списка документов
24   * и каждой пары в этом списке и увеличение счетчика для каждой пары. */
25  HashMap<DocPair, Double> computeIntersections(
26      HashMapList<Integer, Integer> wordToDocs {
27    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
28    Set<Integer> words = wordToDocs.keySet();
29    for (int word : words) {
30      ArrayList<Integer> docs = wordToDocs.get(word);
31      Collections.sort(docs);
32      for (int i = 0; i < docs.size(); i++) {
33        for (int j = i + 1; j < docs.size(); j++) {
34          increment(similarities, docs.get(i), docs.get(j));
35      }
36     }
37    }
38
39    return similarities;
40  }
41
42  /* Увеличение размера пересечения каждой пары документов. */
43  void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
44    DocPair pair = new DocPair(doc1, doc2);
45    if (!similarities.containsKey(pair)) {
46      similarities.put(pair, 1.0);
47    } else {
48      similarities.put(pair, similarities.get(pair) + 1);
49    }
50  }
51
52  /* Вычисление степени сходства по величине пересечения. */
53  void adjustToSimilarities(HashMap<Integer, Document> documents,
54                            HashMap<DocPair, Double> similarities) {
55    for (Entry<DocPair, Double> entry : similarities.entrySet()) {
56      DocPair pair = entry.getKey();
57      Double intersection = entry.getValue();
58      Document doc1 = documents.get(pair.doc1);
59      Document doc2 = documents.get(pair.doc2);
60      double union = (double) doc1.size() + doc2.size() - intersection;
61      entry.setValue(intersection / union);
62    }
63  }
64
65  /* HashMapList<Integer, Integer> связывает Integer
66   * с ArrayList<Integer>. Реализация приведена в приложении. */
```

Для набора документов с минималистическим сходством этот алгоритм будет выполняться намного быстрее исходного «наивного» алгоритма, сравнивающего все пары документов напрямую.

Оптимизированное решение (альтернативное)

Существует альтернативный алгоритм, который могут предложить некоторые кандидаты. Он работает чуть медленнее, но все равно неплохо.

Вспомните более ранний алгоритм, вычислявший степень сходства между двумя документами посредством сортировки. Этот метод можно расширить для целого набора документов.

Представьте, что мы взяли все слова, пометили их идентификатором исходного документа, а затем отсортировали. Приведенный ранее список документов будет выглядеть так:

1_{12}, 1_{13}, 1_{15}, 1_{16}, 2_{14}, 3_{13}, 3_{14}, 4_{14}, 5_{12}, 5_{13}, 5_{15}, 6_{16}, 8_{13}, 8_{15}, 9_{12}, 9_{15}

Далее будем действовать практически так же. Мы перебираем список элементов, и для каждой последовательности идентичных элементов увеличиваем счетчики пересечения для соответствующей пары документов.

Для группировки документов и слов будет использоваться класс `Element`. Первичная сортировка списка проводится по словам, но совпадения разрешаются по идентификатору документа.

```
1  class Element implements Comparable<Element> {
2    public int word, document;
3    public Element(int w, int d) {
4      word = w;
5      document = d;
6    }
7
8    /* Функция для сравнения слов, используемая в ходе сортировки. */
9    public int compareTo(Element e) {
10     if (word == e.word) {
11       return document - e.document;
12     }
13     return word - e.word;
14   }
15 }
16
17 HashMap<DocPair, Double> computeSimilarities(
18     HashMap<Integer, Document> documents) {
19   ArrayList<Element> elements = sortWords(documents);
20   HashMap<DocPair, Double> similarities = computeIntersections(elements);
21   adjustToSimilarities(documents, similarities);
22   return similarities;
23 }
24
25 /* Все слова объединяются в список, сортируемый по словам и документам. */
26 ArrayList<Element> sortWords(HashMap<Integer, Document> docs) {
27   ArrayList<Element> elements = new ArrayList<Element>();
28   for (Document doc : docs.values()) {
29     ArrayList<Integer> words = doc.getWords();
30     for (int word : words) {
31       elements.add(new Element(word, doc.getId()));
32     }
33   }
34   Collections.sort(elements);
35   return elements;
36 }
37
38 /* Увеличение размера пересечения для каждой пары документов. */
39 void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
```

```
40    DocPair pair = new DocPair(doc1, doc2);
41    if (!similarities.containsKey(pair)) {
42      similarities.put(pair, 1.0);
43    } else {
44      similarities.put(pair, similarities.get(pair) + 1);
45    }
46  }
47
48  /* Вычисление сходства на основании пересечения. */
49  HashMap<DocPair, Double> computeIntersections(ArrayList<Element> elements) {
50    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
51
52    for (int i = 0; i < elements.size(); i++) {
53      Element left = elements.get(i);
54      for (int j = i + 1; j < elements.size(); j++) {
55        Element right = elements.get(j);
56        if (left.word != right.word) {
57          break;
58      }
59        increment(similarities, left.document, right.document);
60     }
61    }
62    return similarities;
63  }
64
65  /* Вычисление сходства на основании пересечения. */
66  void adjustToSimilarities(HashMap<Integer, Document> documents,
67                            HashMap<DocPair, Double> similarities) {
68    for (Entry<DocPair, Double> entry : similarities.entrySet()) {
69      DocPair pair = entry.getKey();
70      Double intersection = entry.getValue();
71      Document doc1 = documents.get(pair.doc1);
72      Document doc2 = documents.get(pair.doc2);
73      double union = (double) doc1.size() + doc2.size() - intersection;
74      entry.setValue(intersection / union);
75    }
76  }
```

Первый шаг этого алгоритма медленнее, чем у предыдущего алгоритма, поскольку данные приходится сортировать (вместо простого добавления в список). Второй шаг фактически эквивалентен.

Оба варианта работают намного быстрее исходного «наивного» алгоритма.

XI

Дополнительные материалы

В этом разделе представлены темы, которые обычно выходят за рамки собеседований, но могут встречаться в них время от времени.

Интервьюер не удивится, если вы недостаточно хорошо разбираетесь в этих темах. Займетесь их изучением, если захотите. А если у вас не хватает времени, эти темы не относятся к первоочередным.

Во время подготовки 6-го издания мы обсуждали, какие темы стоит (или не стоит) включать в книгу. Красно-черные деревья? Алгоритм Дейкстры? Топологическая сортировка?

С одной стороны, мне неоднократно предлагали включить эти темы в книгу. Люди настаивали, что эти темы встречаются в собеседованиях «постоянно» (в этом случае мы совершенно по-разному понимаем смысл этого слова!). Очевидно, эти темы были востребованы, по крайней мере, некоторыми. А информация лишней не бывает, верно?

С другой стороны, я знаю, что вопросы по этим темам задаются редко. Конечно, они встречаются; в конце концов, у каждого интервьюера могут быть свои представления о том, что «актуально» и «разумно» для интервью. И все же это бывает редко. Если такой вопрос попадется вам, а тема окажется незнакомой, вряд ли это будет иметь серьезные отрицательные последствия.

Признаюсь: в качестве интервьюера я задавала кандидатам вопросы, в которых решение фактически было результатом применения одного из этих алгоритмов. В тех редких случаях, когда кандидат уже знал алгоритм, особой пользы им эти знания не приносили (впрочем, как и вреда). Я оцениваю умение решать задачи, которые вам ранее не встречались, и поэтому учитываю, был ли используемый алгоритм известен вам заранее.

Мне хотелось бы дать читателю объективное представление о собеседовании, а не запугивать его. Кроме того, я не собираюсь искусственно завышать уровень материала, чтобы книга лучше продавалась за счет вашего времени и сил. Это неправильно и несправедливо по отношению к вам.

Вдобавок я не хочу создавать у интервьюеров (а я знаю, они будут читать эту книгу!) впечатление, будто они должны включать более сложные вопросы в собеседование. Интервьюеры, задавая вопросы по этим темам, вы проверяете знание алгоритмов. Это приведет к тому, что вы откажете многим умным людям.

Однако существует много «пограничных» важных тем. Вопросы по ним задаются нечасто, но иногда они встречаются.

В итоге я решила оставить решение за вами. В конце концов, вам виднее, насколько основательно вы хотите подготовиться к собеседованию. Если вы стремитесь к особо тщательной подготовке — читайте. Если вам нравится изучать структуры

данных и алгоритмы — читайте. Если вы хотите узнать новые подходы к решению задач — читайте.

Но если время поджимает, это не самый важный материал.

Полезные формулы

В этом разделе приведены некоторые формулы, которые могут пригодиться в некоторых вопросах. Формальные доказательства можно найти в Интернете, а мы постараемся объяснить их смысл на интуитивном уровне.

Сумма целых чисел от 1 до *N*

Чему равна сумма 1+2+...+N? Чтобы получить нужную формулу, составим пары из низших и высших значений.

Если `n` четно, то 1 образует пару с `n`, 2 с `n - 1` и т. д. Получаем `n/2` пар, каждая из которых имеет сумму `n + 1`.

Если `n` нечетно, то 0 образует пару с `n`, 1 с `n - 1` и т. д. Получаем `(n+1)/2` пар с суммой `n`.

В любом случае сумма равна `n(n+1)/2`.

Формула часто встречается при анализе вложенных циклов, например:

```
1  for (int i = 0; i < n; i++) {
2    for (int j = i + 1; j < n; j++) {
3      System.out.println(i + j);
4    }
5  }
```

В первой итерации внешнего цикла `for` внутренний цикл выполняется `n - 1` раз. При второй итерации внешнего цикла `for` внутренний цикл выполняется `n - 2` раз, затем `n - 3`, `n - 4` и т. д. Общее количество итераций внутреннего цикла `for` равно `n(n - 1)/2`. Следовательно, время выполнения кода составляет $O(n^2)$.

Сумма степеней 2

Имеется следующая последовательность: $2^0 + 2^1 + 2^2 + ... + 2^n$. Чему равна ее сумма?

Элегантный способ получения искомой формулы основан на двоичной записи этих значений.

	Число	Двоичная запись	Десятичная запись
	2^0	00001	1
	2^1	00010	2
	2^2	00100	4
	2^3	01000	8
	2^4	10000	16
Сумма:	2^5-1	11111	32 – 1 = 31

Следовательно, сумма $2^0 + 2^1 + 2^2 + ... + 2^n$ в двоичной записи представляет последовательность из `(n + 1)` единиц. Результат равен $2^{n+1} - 1$.

Итак, сумма степеней 2 приблизительно равна следующему значению в последовательности.

Основания логарифмов

Допустим, имеется значение $\log_2$ (логарифм по основанию 2). Как преобразовать его в $\log_{10}$? Другими словами, как связаны между собой $\log_b k$ и $\log_x k$?

Займемся вычислениями. Будем считать, что $c = \log_b k$ и $y = \log_x k$.

```
logbk = c --> b^c = k          // Определение логарифма
logx(b^c) = logxk              // Вычислить логарифм обеих сторон b^c = k
c logxb = logxk                // Свойство логарифмов - вынесение экспоненты
c =  logbk = logxk/logxb       // Деление и подстановка c
```

Следовательно, преобразование $\log_2 p$ в $\log_{10}$ осуществляется по следующей формуле:

$$\log_{10} p = \frac{\log_2 p}{\log_2 p}$$

Итог: логарифмы по разным основаниям отличаются только постоянным множителем. По этой причине основание логарифма в выражениях «О большого» обычно не указывается. Оно несущественно, так как константы все равно исключаются из записи.

Перестановки

Сколько существует способов перестановки строки из n уникальных символов? Первый символ выбирается n способами, второй — `n - 1` способами (один вариант уже выбран), третий — `n - 2` способами, и т. д. Следовательно, общее количество перестановок равно n!:

```
n! = n * (n - 1) * (n - 2) * (n - 3) * ... * 1
```

А если из n уникальных символов строится строка длиной k (из уникальных символов)? Рассуждать можно аналогично, только выбор/умножение необходимо прервать ранее.

```
n!/(n-k)! = n * (n - 1) * (n - 2) * (n - 3) * ... * (n - k+1)
```

Сочетания

Имеется множество из n различающихся символов. Сколькими способами можно выбрать новое множество из k символов (без учета порядка)? Иначе говоря, сколько существует подмножеств размером k в множестве из n различных элементов?

Представьте, что мы сначала строим список всех множеств, записывая все строки длины k и исключив дубликаты.

Из раздела «Перестановки» следует, что существуют `n!/(n-k)!` подстрок длиной k.

Так как каждое подмножество размером k может быть упорядочено в `k!` разных строк, каждая строка будет продублирована `k!` раз в этом списке подстрок. Следовательно, для исключения дубликатов нужно разделить общее число подстрок на `k!`.

$$\binom{n}{k} = \frac{1}{k!} * \frac{n1}{(n-k)!} = \frac{n!}{k!(n-k)!}$$

Доказательство методом индукции

Метод индукции доказывает истинность некоторого утверждения. Этот метод тесно связан с рекурсией, а общая схема рассуждений выглядит так.

Требуется доказать, что утверждение `P(k)` истинно для всех `k >= b`.

- **Основание индукции (базовый случай)**: доказываем, что утверждение истинно для `P(b)`. Обычно доказательство сводится к простой подстановке чисел.
- **Предположение**: предполагаем, что утверждение истинно для `P(n)`.
- **Шаг индукции**: доказываем, что если утверждение истинно для `P(n)`, то оно также истинно для `P(n+1)`.

Воспользуемся индукцией для доказательства того, что у множества из `n` элементов существуют 2^n подмножества.

- Определения: пусть `S = {a1, a2, a3, ..., an}` — множество из `n` элементов.
- Основание индукции: доказываем, что у пустого множества `{}` существуют 2^0 подмножества. Утверждение истинно, так как единственным подмножеством `{}` является `{}`.
- Предположим, у `{a1, a2, a3, ..., an}` существуют 2^n подмножества.
- Требуется доказать, что у `{a1, a2, a3, ..., an+1}` существуют 2^{n+1} подмножества.

  Рассмотрим все подмножества `{a1, a2, a3, ..., an+1}`. Ровно половина из них содержит `an+1`, а другая половина не содержит.

Подмножества, не содержащие `an+1`, представляют собой набор подмножеств `{a1, a2, a3, ..., an}`. Согласно предположению их количество равно 2^n.

Так как количество подмножеств, не содержащих `an+1`, равно количеству подмножеств, содержащих `an+1`, существуют 2^n подмножества, содержащих `an+1`.

Следовательно, общее количество подмножеств равно $2^n + 2^n$, то есть 2^{n+1}.

Действительность многих рекурсивных алгоритмов может быть доказана методом индукции.

Топологическая сортировка

Топологической сортировкой направленного графа называется способ упорядочения списка узлов, обладающий следующим свойством: если в графе существует ребро `(a, b)`, то `a` предшествует `b` в списке. Если граф содержит циклы или не является направленным, топологическая сортировка не существует.

У топологической сортировки имеются практические применения. Например, представьте, что граф представляет части на сборочном конвейере. Ребро `(ручка, дверь)` означает, что ручка должна собираться раньше двери. В этом случае топологическая сортировка определяет действительный порядок сборки.

Для построения топологической сортировки можно воспользоваться следующим методом.

1. Найти все узлы, не имеющие входных ребер, и добавить их в список топологической сортировки.
   - Мы знаем, что добавление таких узлов безопасно, так как им не могут предшествовать другие узлы. От них следует избавиться поскорее!
   - Мы знаем, что при отсутствии циклов должен существовать хотя бы один такой узел. В конце концов, можно выбрать произвольный узел и просто отступать назад. Тогда обход либо в какой-то момент остановится (в таком случае был найден узел, не имеющий входных ребер), либо вернется к уже пройденному узлу (и тогда в графе существует цикл).
2. Когда это будет сделано, выходные ребра добавленных узлов исключаются из графа.
   - Такие узлы уже были добавлены в топологическую сортировку, поэтому фактически они не представляют интереса для дальнейшей обработки.
3. Процедура повторяется: алгоритм добавляет узлы, не имеющие входных ребер, и удаляет их выходные ребра. Когда все узлы будут добавлены в список топологической сортировки, работа алгоритма завершается.

Иногда этот алгоритм встречается на собеседованиях. Вероятно, интервьюер не ждет, что вы знаете его заранее. Тем не менее будет полезно уметь вывести его, даже если он никогда не встречался вам ранее.

Алгоритм Дейкстры

В некоторых графах ребрам назначаются весовые коэффициенты. Скажем, если граф представляет города, каждое ребро может представлять дорогу, а вес — время перемещения. В этом случае можно задаться вопросом: как найти самый короткий путь из текущей точки в другую точку `p`? На помощь приходит алгоритм Дейкстры.

Алгоритм Дейкстры предназначен для поиска кратчайшего пути между двумя точками во взвешенном направленном графе (который может содержать циклы). Всем ребрам такого графа должны быть присвоены положительные весовые коэффициенты.

Вместо того чтобы приводить описание алгоритма Дейкстры, попробуем вывести его логическим путем. Возьмем граф, описанный выше. Чтобы найти кратчайший путь из `s` в `t`, можно буквально перебрать все возможные маршруты и сравнить затраченное время.

1. Начать с узла `s`.
2. Для каждого выходного ребра `s` начать обход. Если ребру `(s, x)` назначен вес 5, то для перехода в точку `x` потребуется 5 минут.
3. Каждый раз, когда вы приходите к узлу, проверить, не посещался ли узел ранее. Если узел посещался, обход просто прекращается — текущий путь автоматически хуже, так как один из предшествующих вариантов обхода уже привел к его более раннему посещению. Если же текущий узел еще не посещался, проверить все возможные выходные пути.

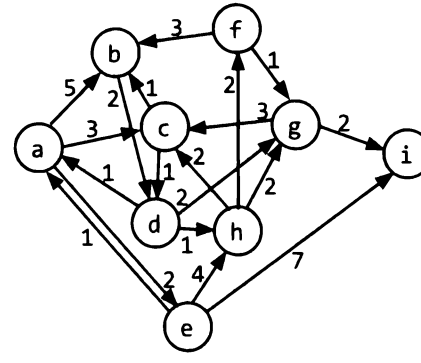
4. Первый путь, по которому мы приходим в `t`, принимается как результат.

Алгоритм работает нормально. Но, конечно, в «настоящем» алгоритме кратчайший путь не будет определяться по часам.

Представьте, что в процессе обхода вы можете немедленно перемещаться из одного узла в соседние узлы (независимо от веса ребра), но время перемещения сохраняется в переменной `time_so_far`. Кроме того, в любой момент времени перемещение ведется только по одному пути, которым всегда является путь с наименьшим значением `time_so_far`. Собственно, приблизительно так и работает алгоритм Дейкстры.

Алгоритм Дейкстры находит путь с минимальным весом от начального узла `s` до *всех* остальных узлов графа.

Рассмотрим следующий граф.



Предположим, мы пытаемся найти кратчайший путь из `a` в `i`. Алгоритм Дейкстры предоставит кратчайшие пути от `a` до всех остальных узлов, среди которых очевидно будет присутствовать и кратчайший путь из `a` в `i`.

Сначала инициализируем несколько переменных:

- `path_weight[node]`: связывает каждый узел с общим весом кратчайшего пути. Все значения инициализируются бесконечностью (кроме значения `path_weight[a]`, которое инициализируется 0).
- `previous[node]`: связывает каждый узел с предыдущим узлом (текущего) кратчайшего пути.
- `remaining`: приоритетная очередь всех узлов в графе, в которой приоритет каждого узла определяется его значением `path_weight`.

После инициализации переменных можно переходить к изменению значений `path_weight`.

(Минимальная) приоритетная очередь представляет собой абстрактный тип данных, который (по крайней мере в данном случае) поддерживает вставку объекта с ключом, удаление объекта с наименьшим ключом и уменьшение ключа. (Представьте обычную очередь, в которой вместо самого старого элемента удаляется элемент с наименьшим или наибольшим приоритетом.) Тип данных назван абстрактным, потому что он определяется своим поведением (то есть операциями), а фактическая реализация может изменяться. Приоритетная очередь может быть реализована на базе массива или минимальной (или максимальной) кучи, а также многих других структур данных.

Алгоритм перебирает узлы `remaining` (пока `remaining` не опустеет) и выполняет следующие действия.

1. Выбрать узел `remaining` с наименьшим значением `path_weight`. Назовем этот узел `n`.
2. Для каждого соседнего узла сравнить `path_weight[x]` (вес текущего кратчайшего пути из `a` в `x`) с `path_weight[n] + edge_weight[(n, x)]`. Другими словами, возможно ли получить путь из `a` в `x` с меньшим весом, если пойти через `n` вместо текущего пути? Если возможно, обновить `path_weight` и `previous`.

3. Удалить `n` из `remaining`.

Когда в `remaining` не остается элементов, `path_weight` содержит вес текущего кратчайшего пути от `a` к каждому узлу. Этот путь строится обратным отслеживанием по `previous`.

Рассмотрим работу алгоритма на примере приведенного выше графа.

1. Первое значение `n` — `a`. Рассматриваем смежные узлы (`b`, `c` и `e`), обновляем значения `path_weight` (5, 3 и 2) и `previous` (2), после чего исключаем `a` из `remaining`.
2. Затем мы переходим к следующему меньшему узлу, которым является `e`. Ранее `path_weight[e]` было присвоено значение 2. Соседними узлами являются `h` и `i`, обновляем `path_weight` (6 и 9) и `previous` для обоих узлов. Заметим, что 6 — это сумма `path_weight[e]` (2) и веса ребра (`e, h`) (4).
3. У следующего меньшего узла `c` значение `path_weight` равно 3. Соседними с ним узлами являются `b` и `d`. Текущее значение `path_weight[d]` равно бесконечности, оно заменяется 4 (`path_weight[c] + weight(edge c, d)`). Значение `path_weight[b]` ранее было задано равным 5. Но поскольку `path_weight[c] + weight(edge c, b)` (3 + 1 = 4) меньше 5, мы заменяем `path_weight[b]` на 4, а в `previous` заносим `c`. Это означает, что путь от `a` к `b` можно улучшить, если пойти через `c`.

Алгоритм продолжает выполняться, пока `remaining` не опустеет. В следующей таблице показаны изменения `path_weight` (слева) и `previous` (справа) на каждом шаге. В верхней строке приведено текущее значение `n` (узел, удаляемый из `remaining`). Строка заполняется черным цветом после удаления из `remaining`.

	INITIAL		n = a		n = e		n = c		n = b		n = d		n = h		n = g		n = f		FINAL	
	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr	wt	pr
a	0	-	удалено																0	-
b	∞	-	5	a			4	c	удалено										4	c
c	∞	-	3	a			удалено												3	a
d	∞	-					4	c			удалено								4	c
e	∞	-	2	a	удалено														2	a
f	∞	-											7	h			удалено		7	h
g	∞	-									6	d			удалено				6	d
h	∞	-			6	e					5	d	удалено						5	d
i	∞	-	∞	-	9	e									8	g			8	g

Когда все будет сделано, для получения фактического пути можно пройти по таблице в обратном направлении, начиная с `i`. В данном случае наименьший вес пути равен 8, и это путь `a -> c -> d -> g -> i`.

Приоритетная очередь и время выполнения

Как упоминалось ранее, в алгоритме используется приоритетная очередь, но эта структура данных может быть реализована по-разному.

Время выполнения алгоритма сильно зависит от реализации приоритетной очереди. Допустим, граф состоит из v вершин и e ребер.

- Если приоритетная очередь реализована на базе массива, `remove_min` будет вызываться до v раз. Каждая операция выполняется за время $O(v)$, поэтому на вызовы `remove_min` будет потрачено время $O(v^2)$. Кроме того, значения `path_weight` и `previous` будут обновляться не более одного раза для каждого ребра, поэтому затраты времени на эти обновления не превышают $O(e)$. Заметим, что e не должно превышать v^2, так как количество ребер не может превышать количество пар вершин. Следовательно, общее время выполнения составит $O(v^2)$.
- Если реализовать приоритетную очередь на базе минимальной кучи, то каждый из вызовов `remove_min` займет время $O(\log v)$ (как и вставка/обновление ключа). Для каждой вершины выполняется один вызов `remove_min`, поэтому общая сложность составит $O(v \log v)$ (v вершин за время $O(\log v)$ каждая). Дополнительно для каждого ребра может быть вызвана одна операция вставки или обновления ключа, что дает $O(e \log v)$. Итоговое время выполнения составляет $O((v+e) \log v)$.

Какой из двух вариантов лучше? Дать однозначный ответ невозможно. Если граф имеет большое количество ребер, то v^2 будет близко к e. В этом случае реализация на базе массива может оказаться предпочтительной, так как $O(v^2)$ будет лучше $O((v + v^2) \log v)$. С другой стороны, для разреженного графа e намного меньше v^2. В этом случае реализация на базе минимальной кучи может оказаться более эффективной.

Разрешение коллизий в хеш-таблице

Практически в любой хеш-таблице возможны коллизии. Проблема может решаться разными способами.

Организация цепочек в формате связного списка

В этом решении (самом распространенном) с элементом массива хеш-таблицы ассоциируется связный список элементов. При относительно небольшом числе коллизий такое решение работает достаточно эффективно.

В худшем случае поиск потребует времени $O(n)$, где n — количество элементов в хеш-таблице. Это возможно только при аномальных данных и/или очень плохой функции хеширования.

Организация цепочек в формате бинарного дерева поиска

Вместо связного списка можно использовать для хранения коллизий бинарное дерево поиска. Тогда время выполнения в худшем случае сокращается до $O(\log n)$.

На практике это решение применяется редко, разве что в случае крайне неравномерного распределения.

Открытая адресация с линейным смещением

В этом случае при возникновении коллизии (то есть если элемент с полученным индексом уже существует) мы просто перемещаем его в следующий индекс массива, пока не будет найдена свободная позиция (или на фиксированное расстояние — допустим, *индекс*+5).

При небольшом количестве коллизий решение получается очень быстрым и эффективным по затратам памяти. Очевидный недостаток заключается в том, что общее количество элементов в хеш-таблице ограничивается размером массива; у решений с цепочками такого недостатка нет.

Также стоит учесть еще одну проблему. Возьмем хеш-таблицу на базе массива размером 100, в которой заполнены индексы с 20 по 29 (и никакие другие). Какова вероятность того, что следующая вставка придется на индекс 30? Она составляет 10%, потому что любому элементу, хешируемому на любой индекс между 20 и 30, в итоге будет сопоставлен индекс 30.

Квадратичное смещение и двойное хеширование

Расстояние между проверяемыми элементами не обязано быть линейным. Например, можно увеличивать расстояние в квадратичной зависимости или же применить вторую хеш-функцию для определения расстояния.

Поиск подстроки по алгоритму Рабина — Карпа

Поиск подстроки `S` в большей строке `B` методом «грубой силы» требует времени $O(s(b-s))$, где `s` — длина `S`, а `b` — длина `B`. Сначала проверяются первые `b - s + 1` символов `B`, после чего для каждого символа следующие `s` символов проверяются на совпадение с `S`.

Алгоритм Рабина — Карпа оптимизирует эту процедуру за счет одного трюка: если две строки совпадают, они должны иметь одинаковые хеш-коды (впрочем, обратное неверно: две разные строки могут иметь одинаковые хеш-коды).

Следовательно, если удастся эффективно заранее вычислить хеш-код каждой последовательности из `s` символов в `B`, потенциальные вхождения `S` можно будет найти за время $O(b)$. После этого остается убедиться в том, что эти вхождения действительно совпадают с `S`.

Например, представьте, что хеш-функция просто вычисляет сумму всех символов (пробел = 0, a = 1, b = 2 и т. д.). Если `S` содержит символы `ear`, а `B = doe are hearing me`, мы просто ищем последовательности с суммой 24 (e + a + r). В строке такие последовательности встречаются трижды. Для каждой последовательности нужно проверить, действительно ли она содержит символы `ear`.

char:	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	24	20	6	19	24	23	13	13	14	24	28	41	30	21	20	18		

Если вычислить все эти суммы, вызывая `hash('doe')`, затем `hash('oe')`, затем `hash('e a')` и т. д., время выполнения все равно составит $O(s(b-s))$.

Вместо этого мы воспользуемся тем фактом, что `hash('oe') = hash('doe') - code('d') + code(' ')`. Вычисление всех хешей занимает время $O(b)$.

Можно возразить, что в худшем случае время выполнения все равно составит $O(s(b-s))$ из-за частого совпадения хеш-кодов. И это совершенно верно — для этой хеш-функции. На практике обычно применяются более совершенная кольцевая хеш-функция. По сути строка `doe` интерпретируется как число по основанию 128 (или количеству символов используемого алфавита).

```
hash('doe') = code('d') * 128² + code('o') * 128¹ + code('e') * 128⁰
```

Эта хеш-функция позволяет удалить `d`, сдвинуть `o` и `e`, а затем добавить пробел.

```
hash('oe ') = (hash('doe') - code('d') * 128²) * 128 + code(' ')
```

Количество ложных совпадений существенно снижается. Хороший выбор хеш-функции обеспечивает ожидаемую временную сложность $O(s + b)$, хотя в худшем случае она составит $O(sb)$.

Вопросы на применение этого алгоритма достаточно часто встречаются на собеседованиях, поэтому будет полезно знать, что подстроки можно идентифицировать за линейное время.

АВЛ-деревья

АВЛ-деревья — один из двух распространенных способов реализации балансировки дерева. Здесь будет рассматриваться только вставка, а информацию об удалении вы при желании сможете найти самостоятельно.

Свойства

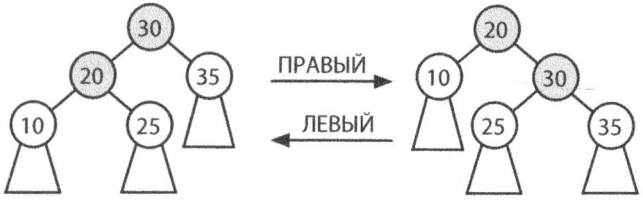
В АВЛ-деревьях в каждом узле хранится высота поддеревьев, корнем которых является данный узел. В этом случае для каждого узла можно проверить сбалансированность по высоте, то есть что высота левого и правого поддеревьев отличается не более чем на 1. Таким образом предотвращаются нарушения в структуре дерева.

```
balance(n) = n.left.height - n.right.height
-1 <= balance(n) <= 1
```

Вставка

При вставке узла баланс некоторых узлов может измениться до –2 или 2. В ходе «раскрутки» стека рекурсии выполняется проверка и исправление баланса в каждом узле. Для этого применяется серия поворотов.

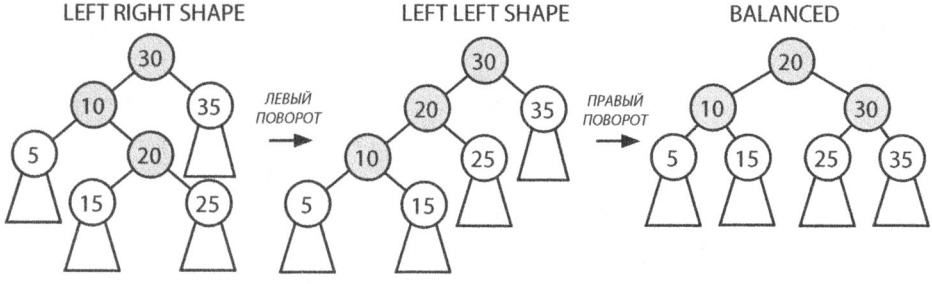
Повороты могут быть левыми или правыми (правый поворот является обратным по отношению к левому).



Способ исправления зависит от значения баланса и места возникновения дисбаланса.

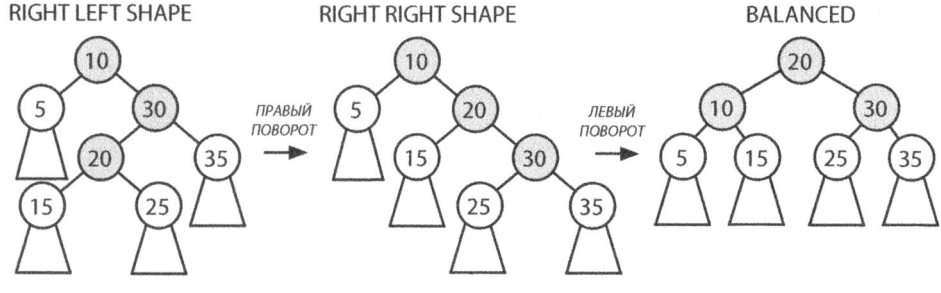
Случай 1. Баланс = 2

В этом случае высота левого поддерева на 2 превышает высоту правого поддерева. Если левая сторона больше, то лишние узлы левого поддерева должны быть обращены налево (LEFT LEFT SHAPE) или направо (LEFT RIGHT SHAPE). Если структура аналогична LEFT RIGHT SHAPE, она посредством поворотов преобразуется в LEFT LEFT SHAPE, а затем в BALANCED. Если структура уже аналогична LEFT LEFT SHAPE, она просто преобразуется в BALANCED.



Случай 2. Баланс = –2

Этот случай является зеркальным отражением предыдущего. Возможны два варианта структуры, RIGHT LEFT SHAPE или RIGHT RIGHT SHAPE. Повороты преобразуют ее в BALANCED.



В обоих случаях под «сбалансированностью» понимается лишь то, что баланс дерева лежит в диапазоне от –1 до 1, а не то, что баланс равен 0.

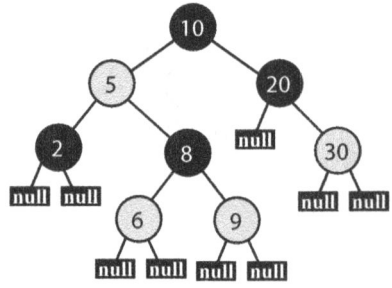
Мы производим рекурсивный обход дерева с исправлением любых дисбалансов. Если в поддереве будет достигнут баланс 0, мы знаем, что все балансы исправлены: эта часть дерева не приведет к тому, что у другого, более высокого поддерева баланс будет равен –2 или 2. При нерекурсивной реализации здесь можно прервать выполнение цикла.

Красно-черные деревья

Красно-черные деревья (разновидность самобалансирующихся бинарных деревьев поиска) не гарантируют строгой сбалансированности, однако и этого достаточно для того, чтобы обеспечить вставку, удаление и выборку за время $O(\log N)$. Они требуют чуть меньшей памяти и перебалансируются быстрее (что означает ускорение вставок и удалений), вследствие чего часто используются в ситуациях с частыми изменениями деревьев.

Работа красно-черных деревьев основана на соблюдении квазичередующейся красно-черной окраски узлов (по определенным правилам, описанным ниже) и требовании о том, чтобы каждый путь от узла к его листьям содержал одинаковое количество черных узлов. Соблюдение этого условия приводит к относительной сбалансированности дерева.

Следующее дерево является красно-черным (красные узлы обозначены серым цветом):



Свойства

1. Каждый узел окрашен либо в красный, либо в черный цвет.
2. Корень окрашен в черный цвет.
3. Листья (NULL-узлы) считаются черными.
4. Каждый красный узел должен иметь два черных дочерних узла. Другими словами, красный узел не может иметь красных дочерних узлов (хотя у черного узла черные дочерние узлы допустимы).
5. Все пути от узла к его листьям должны содержать одинаковое количество черных дочерних узлов.

Причины сбалансированности

Свойство 4 означает, что два красных узла не могут занимать соседние позиции в пути (как родитель и потомок). Следовательно, путь не может содержать более половины красных узлов.

Возьмем два пути от узла (допустим, от корня) к его листьям. Пути должны содержать одинаковое количество черных узлов (свойство 5), поэтому будем считать, что количество красных путей в них различается, насколько это возможно: один путь содержит минимальное, а другой — максимальное количество красных узлов.

Путь 1 (минимум красных узлов): минимальное количество красных узлов равно 0. Следовательно, путь 1 содержит `b` узлов.

Путь 2 (максимум красных узлов): максимальное количество красных узлов равно `b`, так как красные узлы должны иметь черные дочерние узлы, количество которых равно `b`. Следовательно, путь 2 содержит `2b` узлов.

Итак, даже в самом крайнем случае длины путей не могут различаться более чем вдвое. Этого достаточно для того, чтобы обеспечить время выполнения поиска и вставки $O(\log N)$.

Если мы сможем поддерживать эти свойства, дерево будет (относительно) сбалансированным, по крайней мере достаточно сбалансированным для того, чтобы гарантировать время поиска и вставки $O(\log N)$. Вопрос в том, как эффективно обеспечивать поддержание этих свойств. Здесь рассматривается только вставка, а информацию об удалении вы сможете найти самостоятельно.

Вставка

Вставка нового узла в красно-черное дерево начинается с обычной вставки в бинарное дерево поиска.

- Новые узлы вставляются в позициях листьев; это означает, что они замещают черный узел.
- Новые узлы всегда окрашиваются в красный цвет и получают два черных узла (NULL).

Когда это будет сделано, можно переходить к исправлению нарушенных свойств красно-черных деревьев. Возможны два вида нарушений:

- Красные нарушения: красный узел имеет красный дочерний узел (или корень окрашен в красный цвет).
- Черные нарушения: один путь содержит больше черных узлов, чем другой путь.

Вставленный узел окрашен в красный цвет. Количество черных узлов в пути к листу осталось неизменным, поэтому мы знаем, что черных нарушений нет. Тем не менее возможно красное нарушение.

В особом случае, когда в красный цвет окрашен корень, его всегда можно перекрасить в черный цвет для выполнения свойства 2 без нарушения других ограничений.

В остальных случаях наличие красного нарушения означает, что красный узел расположен под другим красным узлом.

Обозначим текущий узел `N`. Его родитель будет обозначаться `P`, его «прародитель» (родитель 2-го уровня) — `G`, а «брат родителя» `P` («дядя» `N`) — `U`. Известно, что:

- `N` и `P` — красные узлы, так как существует красное нарушение.
- Узел `G` определенно черный, так как *до этого* красное нарушение не обнаруживалось.

Неизвестными остаются следующие части:

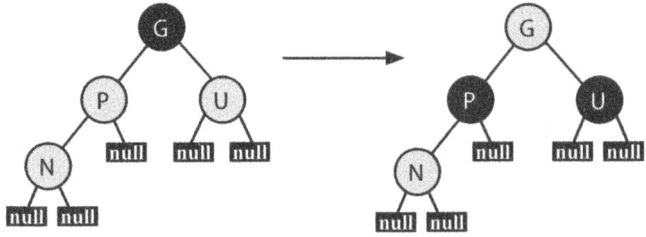
- Узел U может быть как красным, так и черным.
- Узел U может быть как левым, так и правым дочерним узлом.
- Узел N может быть как левым, так и правым дочерним узлом.

Простая комбинаторика показывает, что нам придется рассмотреть 8 случаев. К счастью, некоторые из этих случаев эквивалентны.

Случай 1. Узел U красный

Неважно, является ли U левым или правым дочерним узлом или является ли P левым или правым дочерним узлом. Четыре из восьми случаев можно слить в один.

Если узел U красный, достаточно изменить цвета P, U и G. Перекрашиваем G из черного в красный, а P и U — из красного в черный. Количество черных узлов ни в одном пути не изменилось.



С другой стороны, окрашивание G в красный цвет может создать красное нарушение с родителем G. В таком случае вся логика рекурсивно применяется для обработки красного нарушения, где роль N отводится G.

Учтите, что в обобщенном рекурсивном случае N, P и U также могут иметь поддеревья на месте черных NULL-узлов (листьев на рисунке). В случае 1 эти поддеревья остаются связанными с теми же родителями, так как структура дерева остается неизменной.

Случай 2. Узел U черный

Необходимо рассмотреть возможные конфигурации (левый и правый дочерний узел) N и U. В каждом случае наша цель состоит в исправлении красного нарушения (красный узел над красным) без:

- нарушения упорядочения бинарного дерева поиска;
- создания черного нарушения (один путь содержит больше черных узлов, чем другой).

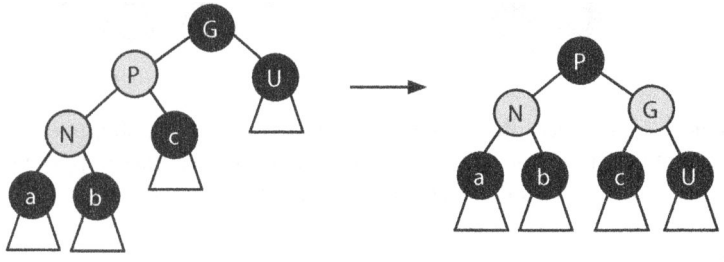
Если это возможно, все хорошо. В каждом из следующих случаев красное нарушение исправляется поворотами, сохраняющими упорядочение узлов.

Кроме того, нижние повороты сохраняют точное количество черных узлов в каждом пути в соответствующей части дерева. Либо NULL-листья являются дочерними узлами поворачивающейся секции, либо поддеревья остаются внутренне неизменными.

Случай А. N и P — левые дочерние узлы

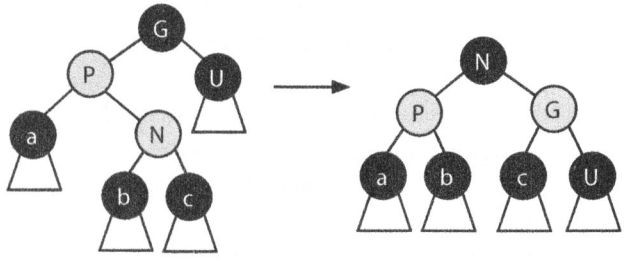
Проблема красного нарушения решается поворотом `N`, `P` и `G` с изменением цветов, описанным ниже.

Если представить симметричный обход, вы увидите, что поворот сохраняет упорядочение узлов (`a <= N <= b <= P <= c <= G <= U`). Дерево поддерживает то же число черных узлов на пути к каждому поддереву `a`, `b`, `c` и `U` (все они могут быть равны NULL).



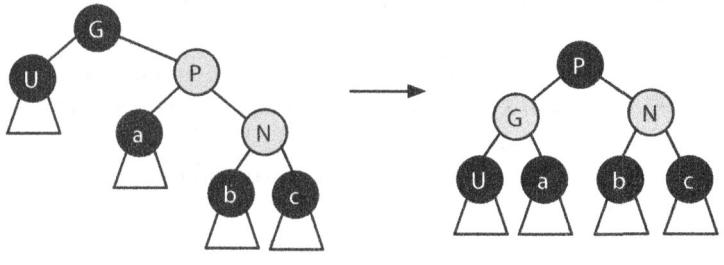
Случай Б. P — левый, а N — правый дочерний узел

Повороты в случае Б решают проблему красного нарушения и сохраняют свойство симметричного обхода: `a <= P <= b <= N <= c <= G <= U`. И снова количество черных узлов остается постоянным на каждом пути до листьев (или поддеревьев).



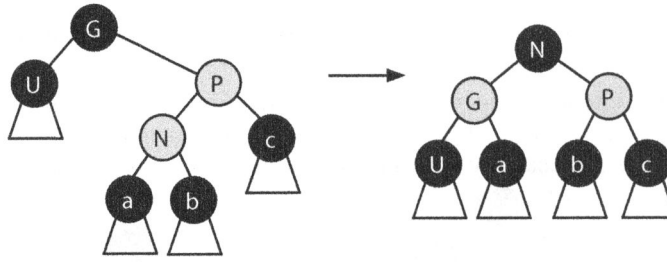
Случай В. N и P — правые дочерние узлы

Этот случай является зеркальным по отношению к случаю А.



Случай Г. N — левый, а P — правый дочерний узел

В каждом из подслучаев случая 2 средний элемент по значению `N`, `P` и `G` поворачивается, чтобы стать корнем того, что было поддеревом `G`, после чего этот элемент и `G` меняются цветами.



Не пытайтесь просто запомнить эти случаи. Вместо этого разберитесь, как они работают. Как в каждом случае обеспечить отсутствие красных нарушений, черных нарушений и нарушений свойства бинарного дерева поиска?

MapReduce

Технология MapReduce широко используется при проектировании систем, обрабатывающих большие объемы данных. Как следует из названия, программа MapReduce требует написания двух шагов: отображения (Map) и свертки (Reduce). Все остальное делает система.

1. Система распределяет данные между машинами.
2. Каждая машина начинает выполнять программу Map, предоставленную пользователем.
3. Программа Map получает данные и выдает пару <*ключ, значение*>.
4. Процесс Shuffle, предоставляемый системой, реорганизует данные так, что все пары <*ключ, значение*>, связанные с заданным ключом, попадают на одну машину для обработки Reduce.
5. Предоставленная пользователем программа Reduce получает ключ и набор значений и каким-то образом выполняет «свертку», генерируя новый ключ и значение. Результат может быть передан обратно программе Reduce для дальнейшей свертки.

Типичный пример использования MapReduce — своего рода программа «Hello World» для MapReduce — подсчитывает частоту вхождения слов в наборе документов.

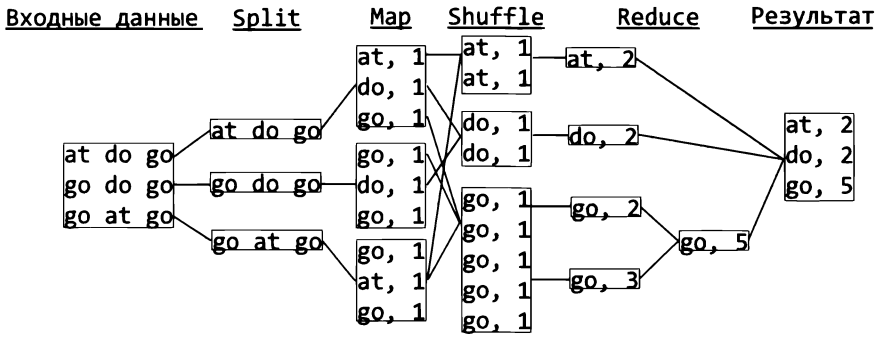
Конечно, программу можно оформить в виде одной функции, которая читает все данные, подсчитывает количество вхождений каждого слова в хеш-таблице и выводит результат.

MapReduce позволяет организовать параллельную обработку документов. Функция Map читает документ и выводит только каждое слово и счетчик (который всегда равен 1). Функция Reduce читает ключи (слова) и связанные с ними значения (счетчики), после чего выводит сумму счетчиков. Сумма также может стать вводом для другого вызова Reduce по тому же ключу (как показано на диаграмме).

```
1  void map(String name, String document):
2    for each word w in document:
3      emit(w, 1)
```

```
4
5  void reduce(String word, Iterator partialCounts):
6    int sum = 0
7    for each count in partialCounts:
8      sum += count
9    emit(word, sum)
```

Следующая диаграмма показывает, как работает MarReduce в конкретном примере.



Дополнительные материалы

Итак, вы освоили материал книги и хотите узнать больше? Хорошо. Ниже перечислены некоторые темы, с которых можно начать:

- **Алгоритм Беллмана — Форда**: поиск кратчайших путей из одного узла во взвешенном направленном графе с положительными и отрицательными весами ребер.
- **Алгоритм Флойда — Уоршелла**: поиск кратчайших путей во взвешенном графе с положительными или отрицательными весами ребер (но без циклов с отрицательным весом).
- **Минимальные остовные деревья**: во взвешенном связном ненаправленном графе «остовным деревом» называется дерево, соединяющее все вершины. Минимальным остовным деревом называется остовное дерево с минимальным весом. Существуют разные алгоритмы для решения этой задачи.
- **B-деревья**: самобалансирующиеся деревья (не бинарные деревья поиска!), обычно применяемые для хранения информации на дисках или других устройствах. Похожи на красно-черные деревья, но используют меньшее количество операций ввода/вывода.
- **A***: поиск пути с наименьшей стоимостью между исходным и целевым узлом (или одним из нескольких целевых узлов). Алгоритм является расширением алгоритма Дейкстры и обеспечивает лучшую производительность за счет применения эвристики.
- **Интервальные деревья**: расширенная версия сбалансированного бинарного дерева поиска, в которой вместо простых значений хранятся интервалы. Например, гостиница может использовать эту структуру данных для хранения списка предварительно заказанных номеров и эффективного получения информации о том, кто будет оставаться в гостинице в заданный промежуток времени.

- **Раскраска графов**: способ назначения цветов вершинам графа, при котором никакие две смежные вершины не имеют одинакового цвета. Существуют различные алгоритмы такого рода, например проверки возможности раскраски графа с использованием только k цветов.
- **P, NP и NP-полнота**: эти термины используются для описания классов задач. P-задачи решаются быстро (за полиномиальное время). К категории NP относятся задачи, в которых при наличии решения можно быстро проверить это решение. NP-полные задачи составляют подмножество NP-задач, которые могут сводиться друг к другу (то есть если вы нашли решение одной задачи, то модификация этого решения позволит решить другие задачи за полиномиальное время).

  Существует очень известный (и остающийся открытым) вопрос об эквивалентности P = NP, но ученые склонны считать, что ответ на него отрицателен.
- **Комбинаторика и теория вероятностей**: в этой области есть немало тем для изучения: случайные переменные, средние значения и биномиальные коэффициенты.
- **Двудольные графы**: «двудольным» называется граф, узлы которого можно разбить на два множества таким образом, что концы каждого ребра принадлежат двум разным множествам. Существует алгоритм для проверки того, является ли заданный граф двудольным. Заметим, что двудольный граф эквивалентен графу, который может быть раскрашен в два цвета.
- **Регулярные выражения**: вы должны знать, что такое регулярные выражения и для чего они нужны (хотя бы в общих чертах). Также стоит узнать, как работают алгоритмы поиска совпадений для регулярных выражений, и ознакомиться с базовым синтаксисом регулярных выражений.

XII

Библиотека кода

HashMapList<T, E>

Класс `HashMapList` фактически представляет собой сокращение для `HashMap<T, ArrayList<E>>`. Он позволяет связать элемент типа `T` с `ArrayList` типа `E`.

Допустим, вам понадобилась структура данных, связывающая целое число со списком строк. Обычно пришлось бы написать нечто вроде:

```
1  HashMap<Integer, ArrayList<String>> maplist =
2    new HashMap<Integer, ArrayList<String>>();
3  for (String s : strings) {
4    int key = computeValue(s);
5    if (!maplist.containsKey(key)) {
6      maplist.put(key, new ArrayList<String>());
7    }
8    maplist.get(key).add(s);
9  }
```

С `HashMapList` аналогичный фрагмент будет выглядеть так:

```
1  HashMapList<Integer, String> maplist = new HashMapList<Integer, String>();
2  for (String s : strings) {
3    int key = computeValue(s);
4    maplist.put(key, s);
5  }
```

Изменение не столь серьезное, но оно упрощает код.

```
1  public class HashMapList<T, E> {
2    private HashMap<T, ArrayList<E>> map = new HashMap<T, ArrayList<E>>();
3
4    /* Вставка элемента в список с заданным ключом. */
5    public void put(T key, E item) {
6      if (!map.containsKey(key)) {
7        map.put(key, new ArrayList<E>());
8     }
9      map.get(key).add(item);
10   }
11
12   /* Вставка списка элементов с заданным ключом. */
13   public void put(T key, ArrayList<E> items) {
14     map.put(key, items);
15   }
16
17   /* Чтение списка с заданным ключом. */
18   public ArrayList<E> get(T key) {
19     return map.get(key);
20   }
```

```
21
22    /* Проверка существования ключа в коллекции. */
23    public boolean containsKey(T key) {
24      return map.containsKey(key);
25    }
26
27    /* Проверка наличия значений в списке для заданного ключа. */
28    public boolean containsKeyValue(T key, E value) {
29      ArrayList<E> list = get(key);
30      if (list == null) return false;
31      return list.contains(value);
32    }
33
34    /* Получение списка ключей. */
35    public Set<T> keySet() {
36      return map.keySet();
37    }
38
39    @Override
40    public String toString() {
41      return map.toString();
42    }
43  }
```

TreeNode (бинарное дерево поиска)

Ничто не мешает использовать встроенный класс бинарного дерева (более того, это даже желательно) там, где это возможно. Тем не менее возможно это не всегда. Во многих задачах необходим доступ к внутренней структуре класса узла или дерева, а это исключает возможность использования встроенных библиотек.

Класс `TreeNode` предоставляет разнообразную функциональность, большая часть которой вряд ли понадобится во всех вопросах/решениях. Например, класс `TreeNode` хранит ссылку на родительский узел, хотя часто эта информация не нужна (или ее использование запрещено условиями задачи).

Для простоты в реализации дерева данные хранятся в формате целых чисел.

```
1  public class TreeNode {
2    public int data;
3    public TreeNode left, right, parent;
4    private int size = 0;
5
6    public TreeNode(int d) {
7      data = d;
8      size = 1;
9    }
10
11    public void insertInOrder(int d) {
12      if (d <= data) {
13        if (left == null) {
14          setLeftChild(new TreeNode(d));
15        } else {
16          left.insertInOrder(d);
17      }
18      } else {
```

```
19        if (right == null) {
20          setRightChild(new TreeNode(d));
21        } else {
22          right.insertInOrder(d);
23       }
24      }
25       size++;
26     }
27
28     public int size() {
29       return size;
30     }
31
32     public TreeNode find(int d) {
33       if (d == data) {
34         return this;
35       } else if (d <= data) {
36         return left != null ? left.find(d) : null;
37       } else if (d > data) {
38         return right != null ? right.find(d) : null;
39      }
40       return null;
41     }
42
43     public void setLeftChild(TreeNode left) {
44       this.left = left;
45       if (left != null) {
46         left.parent = this;
47      }
48     }
49
50     public void setRightChild(TreeNode right) {
51       this.right = right;
52       if (right != null) {
53         right.parent = this;
54      }
55     }
56
57   }
```

Дерево реализовано как бинарное дерево поиска, однако оно также может использоваться для других целей. Для этого достаточно использовать методы `setLeftChild`/`setRightChild` или переменные `left` и `right`. По этой причине методы и переменные были объявлены открытыми. Такой уровень доступа оказывается полезным во многих задачах.

LinkedListNode (связный список)

Как и в случае с классом `TreeNode`, нам часто требуется доступ к внутренней реализации связного списка, не поддерживаемый встроенным классом. По этой причине мы написали собственную реализацию связного списка и использовали ее во многих задачах.

```
1  public class LinkedListNode {
2    public LinkedListNode next, prev, last;
```

```
3    public int data;
4    public LinkedListNode(int d, LinkedListNode n, LinkedListNode p){
5      data = d;
6      setNext(n);
7      setPrevious(p);
8    }
9
10    public LinkedListNode(int d) {
11      data = d;
12    }
13
14    public LinkedListNode() { }
15
16    public void setNext(LinkedListNode n) {
17      next = n;
18      if (this == last) {
19        last = n;
20     }
21      if (n != null && n.prev != this) {
22        n.setPrevious(this);
23     }
24    }
25
26    public void setPrevious(LinkedListNode p) {
27      prev = p;
28      if (p != null && p.next != this) {
29        p.setNext(this);
30     }
31    }
32
33    public LinkedListNode clone() {
34      LinkedListNode next2 = null;
35      if (next != null) {
36        next2 = next.clone();
37     }
38      LinkedListNode head2 = new LinkedListNode(data, next2, null);
39      return head2;
40    }
41  }
```

В этом случае методы и переменные также были объявлены открытыми, потому что такой доступ часто необходим по условиям задачи. Хотя такая функциональность нарушает инкапсуляцию данных, она нужна для наших целей.

Trie и TrieNode

Структура данных нагруженного дерева используется в нескольких задачах для определения того, является ли слово префиксом других слов в словаре (или списке допустимых слов). Также она часто применяется при рекурсивном построении слов, чтобы для недействительных слов обработку можно было прервать заранее.

```
1  public class Trie {
2    // Корень нагруженного дерева
3    private TrieNode root;
4
```

```
5    /* Получает список строк в аргументе и строит нагруженное
6     * дерево, содержащее эти строки. */
7    public Trie(ArrayList<String> list) {
8      root = new TrieNode();
9      for (String word : list) {
10       root.addWord(word);
11     }
12   }
13
14
15   /* Получает список строк в аргументе и строит нагруженное
16    * дерево, содержащее эти строки. */
17   public Trie(String[] list) {
18     root = new TrieNode();
19     for (String word : list) {
20       root.addWord(word);
21     }
22   }
23
24   /* Проверяет, содержит ли нагруженное дерево строку с префиксом,
25    * переданным в аргументе. */
26   public boolean contains(String prefix, boolean exact) {
27     TrieNode lastNode = root;
28     int i = 0;
29     for (i = 0; i < prefix.length(); i++) {
30       lastNode = lastNode.getChild(prefix.charAt(i));
31       if (lastNode == null) {
32         return false;
33       }
34     }
35     return !exact || lastNode.terminates();
36   }
37
38   public boolean contains(String prefix) {
39     return contains(prefix, false);
40   }
41
42   public TrieNode getRoot() {
43     return root;
44   }
45 }
```

Класс `Trie` использует класс `TrieNode`, реализация которого приведена ниже.

```
1  public class TrieNode {
2    /* Дочерние узлы этого узла в нагруженном дереве.*/
3    private HashMap<Character, TrieNode> children;
4    private boolean terminates = false;
5
6    /* Данные узла, то есть хранящийся в нем символ.*/
7    private char character;
8
9    /* Строит пустой узел и инициализирует список его дочерних узлов.
10     * Используется только для создания корневого узла. */
11   public TrieNode() {
12     children = new HashMap<Character, TrieNode>();
13   }
```

```
14
15    /* Строит узел и сохраняет символ в качестве его значения.
16     * Инициализирует список дочерних узлов пустым контейнером HashMap. */
17    public TrieNode(char character) {
18      this();
19      this.character = character;
20    }
21
22    /* Возвращает символ, хранящийся в узле. */
23    public char getChar() {
24      return character;
25    }
26
27    /* Добавляет слово в нагруженное дерево и проводит
28     * рекурсивное создание дочерних узлов. */
29    public void addWord(String word) {
30      if (word == null || word.isEmpty()) {
31        return;
32     }
33
34      char firstChar = word.charAt(0);
35
36      TrieNode child = getChild(firstChar);
37      if (child == null) {
38        child = new TrieNode(firstChar);
39        children.put(firstChar, child);
40     }
41
42      if (word.length() > 1) {
43        child.addWord(word.substring(1));
44      } else {
45        child.setTerminates(true);
46     }
47    }
48
49    /* Находит дочерний узел, значением которого является переданный
50     * символ. Если такого дочернего узла нет, возвращается null. */
51    public TrieNode getChild(char c) {
52      return children.get(c);
53    }
54
55    /* Сообщает, представляет ли узел конец полного слова. */
56    public boolean terminates() {
57      return terminates;
58    }
59
60    /* Сохраняет признак конца полного слова.*/
61    public void setTerminates(boolean t) {
62      terminates = t;
63    }
64  }
```

БИБЛИОТЕКА ПРОГРАММИСТА

БЕСТСЕЛЛЕР AMAZON

Очередное собеседование обернулось разочарованием... в очередной раз. Никто из десяти кандидатов не получил работу. Может быть, «экзаменаторы» были слишком строги?

Увы, для поступления на работу в ведущую IT-компанию академического образования недостаточно. Учебники — это замечательно, но они не помогут вам пройти собеседование, для этого нужно готовиться на реальных вопросах. Нужно решать реальные задачи и изучать встречающиеся закономерности. Главное — разработка новых алгоритмов, а не запоминание существующих задач.

«Карьера программиста» основана на опыте практического участия автора во множестве собеседований, проводимых лучшими компаниями. Это квинтэссенция сотен интервью со множеством кандидатов, результат ответов на тысячи вопросов, задаваемых кандидатами и интервьюерами в ведущих мировых корпорациях.

Используя материал этой книги, вы с легкостью подготовитесь к устройству на работу в Google, Microsoft или любую другую ведущую IT-компанию:

- *Тонкости прохождения собеседований.*
- *Свежий подход к академическим знаниям.*
- *Алгоритмизация, программирование, дизайн.*
- *189 тестовых заданий из собеседований в крупнейших IT-компаниях.*

Шестое издание этого мирового бестселлера поможет вам наилучшим образом подготовиться к собеседованию при приеме на работу программистом или руководителем в крупную IT-организацию или перспективный стартап. Основную часть книги составляют ответы на технические вопросы и задания, которые обычно получают соискатели на собеседовании в таких компаниях, как Google, Microsoft, Apple, Amazon и других. Рассмотрены типичные ошибки, а также эффективные методики подготовки к собеседованию.

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/**piterbooks**
youtube.com/**ThePiterBooks**
vk.com/**piterbooks**
facebook.com/**piterbooks**

ISBN: 978-5-496-03239-1