

Г. Лакман Макдауэлл

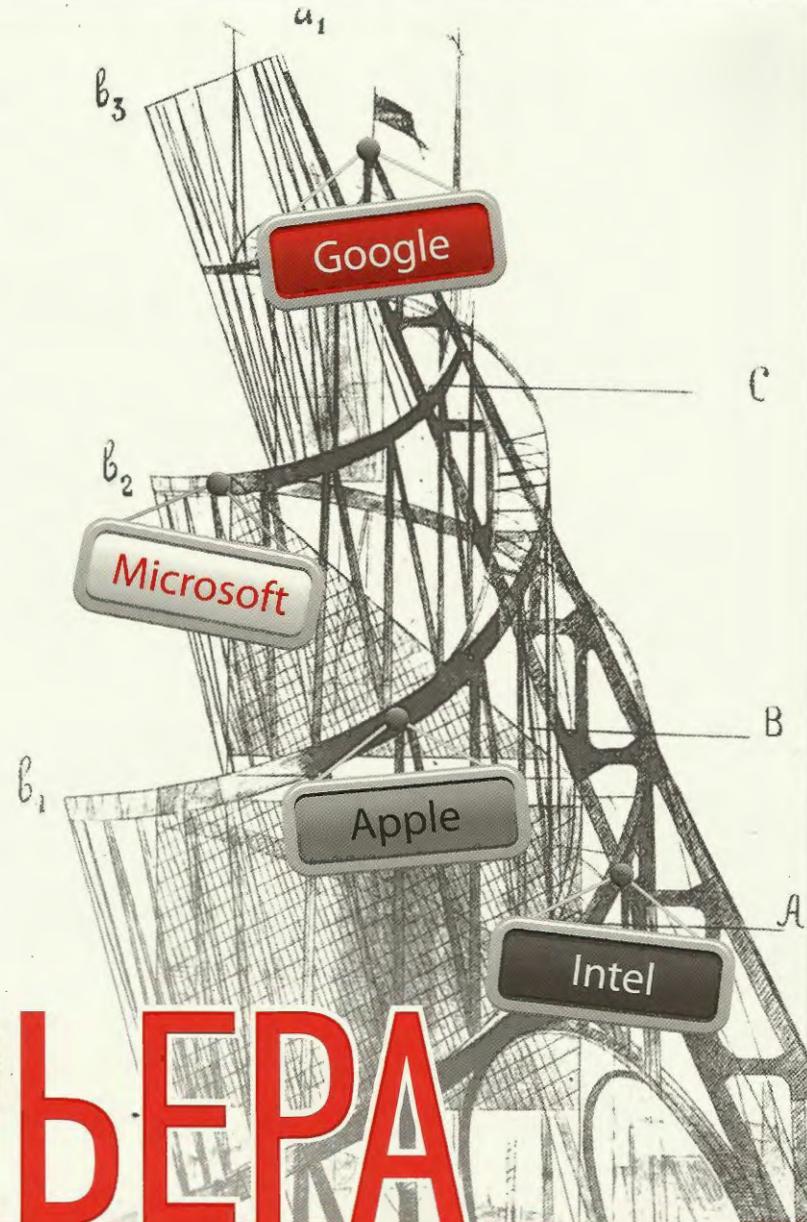
150



тестовых
заданий

КАРЬЕРА ПРОГРАММИСТА

Как устроиться на работу в Google,
Microsoft или другую ведущую IT-компанию



БЕЛЫЙ КОВРЫ
ЧИСЛО ПРО
ВПЛ

Г. Лакман Макдауэлл

КАРЬЕРА ПРОГРАММИСТА

Как устроиться на работу в Google,
Microsoft или другую ведущую IT-компанию

150
тестовых
заданий



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2012

ББК 32.973.2

УДК 004.3

Л19

- Лакман Макдауэлл Г.
Л19 Карьера программиста. Как устроиться на работу в Google, Microsoft или другую ведущую IT-компанию. — СПб.: Питер, 2012. — 416 с.: ил.

ISBN 978-5-459-01120-3

Пятое 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.

ББК 32.973.2

УДК 004.3

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1451578270 англ.
ISBN 978-5-459-01120-3

© 2008–2011 by Gayle Laakmann McDowell

© Перевод на русский язык ООО Издательство «Питер», 2012

© Издание на русском языке, оформление ООО Издательство «Питер», 2012

Часть III. Нестандартные случаи	31
Часть IV. Перед собеседованием	36
Часть V. Подготовка к поведенческим вопросам	40
Часть VI. Технические вопросы	44
Часть VII. Жизнь после собеседования	56
Часть VIII. Вопросы собеседования	61
Часть IX. Решения	140
Благодарности	412
Об авторе	413

Оглавление

Предисловие.....	13
Введение.....	14
Что-то не так.....	14
Мой подход.....	15
Моя страсть	15
От издательства.....	15
Часть I. Процесс собеседования	16
Небольшое вступление.....	16
Как выбираются вопросы	16
График и карта подготовки.....	17
Процедура оценки	20
Неправильные ответы.....	20
Пресс-код	21
10 наиболее частых ошибок	21
1. Использование компьютера	21
2. Игнорирование поведенческих вопросов	22
3. Отказ от псевдо-интервью.....	22
4. Попытка зазубрить ответ	22
5. Решение задачи «в уме»	22
6. Спешка	22
7. Грязный код.....	23
8. Отказ от проверки	23
9. Небрежное отношение к исправлению ошибок.....	23
10. Отказ от решения	23
Часто задаваемые вопросы.....	23
Нужно ли мне говорить интервьюеру, что я уже знаком с вопросом?	23
Какой язык программирования следует использовать?	24
После собеседования мне ничего не сказали. Мне отказали?	24
Могу ли я попытаться еще раз, если мне отказали?	24
Часть II. За кулисами.....	25
Microsoft.....	25
Amazon.....	26
Google.....	27
Apple.....	28
Facebook.....	29
Yahoo!.....	30

Часть III. Нестандартные случаи	31
Кандидат-профессионал	31
Тестеры и SDET	31
Совет	32
Менеджеры программ и менеджеры продукта	32
Ведущие разработчики и менеджеры	33
Стартап	34
Процесс подачи заявления	34
Виза и разрешение на работу	34
Резюме	35
Процесс собеседования	35
Часть IV. Перед собеседованием	36
Получаем «правильный» опыт	36
Налаживаем связи	37
Правильный круг знакомств	37
Как построить сильный круг знакомств	37
Идеальное резюме	38
Правильный размер	38
Трудовой стаж	38
Проекты	39
Языки программирования и программные продукты	39
Часть V. Подготовка к поведенческим вопросам	40
Поведенческие вопросы	40
Как подготовиться	40
Ваши слабые места	41
Что заставляет вас работать	41
Какие вопросы нужно задавать интервьюеру	41
Ответы на поведенческие вопросы	42
Отвечайте четко, но без высокомерия	42
Сократите подробности до минимума	42
Структурируйте ответ	42
Часть VI. Технические вопросы	44
Подготовка	44
Как организовать подготовку	44
Что нужно знать	44
Таблица степеней двойки	45
Нужно ли знать все о программировании на C++, Java или других языках	45
Ответы на технические вопросы	46
Пять шагов к решению	46

Пять подходов к алгоритмизации.....	4
Подход 1. Приводим пример.....	4
Подход 2. Сопоставление с образцом	4
Подход 3. Упростить и обобщить.....	4
Подход 4. Базовый случай и сборка решения.....	5
Подход 5. Мозговой штурм структур данных.....	5
Как выглядит хороший код.....	5
Структуры данных	5
Обоснованное многократное использование кода.....	5
Модульность.....	5
Гибкость и надежность.....	5
Проверка	5
Часть VII. Жизнь после собеседования.....	50
Реакция на предложение и на отказ.....	50
Сроки принятия решения.....	50
Вы отказываетесь от работы	50
Вам отказали.....	50
Вам сделали предложение.....	50
Финансовый пакет	50
Карьерный рост	50
Стабильность компаний.....	50
Удовольствие от работы	50
Переговоры	50
На работе	50
Создайте график своего карьерного роста.....	50
Устанавливайте прочные отношения	50
Спросите себя, что вам нужно.....	50
Часть VIII. Вопросы собеседования	50
1. Массивы и строки.....	50
Хэш-таблицы.....	50
ArrayList (динамический массив)	50
StringBuffer (буфер строк)	50
Вопросы интервью	50
2. Связные списки	50
Создание связного списка	50
Удаление узла из односвязного списка	50
Метод бегунка.....	50

Рекурсия и связные списки	65
Вопросы собеседования	65
3. Стек и очередь.....	67
Реализация стека	67
Реализация очереди.....	67
Вопросы собеседования	68
4. Деревья и графы.....	70
Потенциальные ловушки.....	70
Бинарное дерево vs бинарное дерево поиска	70
Сбалансировано vs несбалансировано	70
Полнота дерева	70
Обход бинарного дерева.....	70
Балансировка: красно-черные и АВЛ-деревья	70
Префиксное дерево	71
Обход графа.....	71
Поиск в глубину	71
Поиск в ширину	72
Вопросы собеседования	72
5. Поразрядная обработка.....	74
Расчеты на бумаге.....	74
Биты: трюки и факты	75
Основные задачи: получение, установка, очистка и обновление бита	75
Извлечение бита.....	75
Установка бита	75
Очистка бита	75
Обновление бита	76
Вопросы собеседования	76
6. Головоломки.....	78
Начните говорить	78
Правила и шаблоны	78
Балансировка худшего случая	79
Алгоритмический подход	79
Вопросы собеседования	80
7. Математика и теория вероятностей	81
Простые числа	81
Делимость.....	81
Является ли число простым	81
Список простых чисел: решето Эратосфена	82

Теория вероятностей.....	83
Вероятность события {A and B}	83
Вероятность события {A or B}	84
Независимость событий.....	84
Взаимоисключающие события	84
Обратите внимание!	85
Вопросы собеседования	85
8. Объектно-ориентированное проектирование	86
Как подготовиться к заданиям по ООП	86
Разработка шаблонов	87
Singleton	87
Factory Method	88
Вопросы собеседования	88
9. Рекурсия и динамическое программирование	90
С чего начать	90
Динамическое программирование	90
Простой пример динамического программирования: числа Фибоначчи	90
Рекурсивные и итерационные решения	91
Вопросы собеседования	92
10. Сортировка и поиск.....	94
Общие алгоритмы сортировки	94
Алгоритмы поиска.....	97
Вопросы собеседования	97
11. Масштабируемость и ограничения памяти.....	99
Пошаговый подход.....	99
Что нужно знать: информация, стратегия и проблема.....	99
Типичная система	99
Разделение данных.....	100
Пример: найдите все документы, содержащие список слов	101
Вопросы собеседования	102
12. Тестирование.....	104
Чего ожидает интервьюер.....	104
Тестирование реального объекта	105
Тестирование программного обеспечения	106
Тестирование функций	107
Поиск и устранение неисправностей.....	108
Вопросы собеседования	109

13. С и С++	110
Классы и наследование.....	110
Конструкторы и деструкторы	111
Виртуальные функции	111
Виртуальный деструктор.....	112
Значения по умолчанию	113
Перезагрузка операторов.....	114
Указатели и ссылки	114
Ссылки.....	114
Арифметика указателей	114
Шаблоны	115
Вопросы собеседования	116
14. Java.....	117
Подход к изучению	117
Ключевое слово final.....	117
Ключевое слово finally	117
Метод finalize.....	118
Перегрузка vs переопределение	119
Java Collection Framework	120
Вопросы собеседования	120
15. Базы данных.....	122
SQL-синтаксис и его варианты.....	122
Денормализованные и нормализованные базы данных	122
SQL-операторы	123
Запрос 1: регистрация студента.....	123
Запрос 2: размер аудитории	124
Проектирование небольшой базы данных	125
Проектирование больших баз данных	126
Вопросы собеседования	126
16. Потоки и блокировки	128
Потоки в Java	128
Расширение класса Thread.....	129
Расширение класса Thread vs реализация Runnable-интерфейса.....	130
Синхронизация и блокировки	130
Методы синхронизации.....	130
Синхронизированные блоки кода	132
Блокировки	132
Взаимные блокировки и их предотвращение.....	133
Вопросы собеседования	134

17. Задачи умеренной сложности.....	135
18. Задачи повышенной сложности	138
Часть IX. Решения.....	140
1. Массивы и строки.....	140
2. Связные списки.....	150
3. Стеки и очереди.....	166
4. Деревья и графы.....	182
5. Поразрядная обработка.....	200
6. Головоломки.....	214
7. Математика и теория вероятностей.....	219
8. Объектно-ориентированное проектирование.....	233
9. Рекурсия и динамическое программирование	264
10. Сортировка и поиск	285
11. Масштабируемость и ограничения памяти.....	301
12. Тестирование.....	316
13. С и C++	321
14. Java.....	332
15. Базы данных	339
16. Потоки и блокировки	345
17. Задачи умеренной сложности	357
18. Задачи повышенной сложности	385
Благодарности.....	412
Об авторе	413

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 для компании Google, я выбирала из множества кандидатов самого подходящего. Именно поэтому я с полной уверенностью могу утверждать, что мне знакомы все круги того зла, который называется устройством на работу.

Если вы читаете эти строки, то это означает, что вы собираетесь пройти собеседование завтра, на следующей неделе или через год. И смею предположить, это собеседование будет иметь отношение к информационным технологиям.

В этой книге мы не будем заниматься основами, вы не найдете здесь информации о том, что такое бинарное дерево поиска или как пройтись по связанному списку. Эти элементарные вещи должны быть вам знакомы, а если это не так, то эта книга не для вас.

Моя задача — помочь вам перейти в IT-области на новый уровень и понять, как использовать свои знания, чтобы с успехом пройти собеседование.

В пятом издании моей книги вас ожидают более 400 страниц с вопросами, задачами, решениями и пр. Обязательно загляните на сайт www.careercup.com, там вы можете общаться с другими соискателями и получить новую информацию.

Хорошая подготовка позволит вам расширить ваши технические и коммуникативные способности, а это никогда не бывает лишним.

Даже если информация во вступительных главах вам покажется неинтересной или хорошо знакомой, не пролистывайте их: может быть, именно там найдется тот самый ключик, который откроет для вас заветную дверь офиса потенциального работодателя.

Не расслабляйтесь — собеседование будет сложным! В свое время (в период моей Google'ской работы) я видела многих интервьюеров, одни из них задавали легкие вопросы, а другие — сложные. Но сложность вопросов никак не влияла на результат собеседования.

Главное — не вопросы, а ваши ответы на них. И ваши ответы должны выгодно отличаться от ответов других кандидатов. И не паникуйте, если вам достался сложный вопрос, — те, кто его задают, знают, что вопрос сложен и не ждут от вас моментального ответа.

Читайте, учитесь, а я желаю вам удачи!

Гэйл Лакман Макдауэлл,
основатель CareerCup.com

Введение

Что-то не так

Мы провели очередное собеседование... И никто из десяти кандидатов не получил работу. Может быть, мы были слишком строги?

Лично меня ждало глубокое разочарование: мой бывший студент не прошел собеседование, а ведь я его рекомендовала. У него был достаточно высокий средний балл (3,73 GPA¹) в Вашингтонском университете — одной из лучших школ мира по компьютерным дисциплинам, — и он активно занимался OpenSource-проектами. Он был энергичен, креативен, он упорно трудился и был компьютерным фанатом в хорошем смысле этого слова.

Но, к сожалению, члены комиссии были неумолимы, и я должна была согласиться с их мнением. Даже если бы сыграла свою роль моя рекомендация, моему ученику отказали бы на более поздних этапах отбора. Слишком много было «красных» карточек.

Интервьюеры ожидали, что он проявит креативность мышления, а он просто изо всех сил решал задачи, поставленные на собеседовании. 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 смог оптимизировать решения для других сценариев. Когда дело дошло до написания кода, он допустил множество ошибок. Это был не худший кандидат, но все видели, как он далек от победы.

Пару недель спустя он подошел ко 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 только знала.

Он тщательно готовился, как и большинство кандидатов. Он изучил K&R² и CLRS³. Он может описать в подробностях множество способов балансировки дерева и умеет делать на C такие вещи, которые большинству программистов просто не снились.

Мне пришлось сказать ему горькую правду — книжного академического образования недостаточно. Книги — это замечательно, но они не помогут вам пройти собеседование. Почему? Подскажу: интервьюеры не встречали «деревьев» со времен обучения в университете.

¹ GPA (Grade Point Average) — средний балл за время обучения. В США вместо пятибалльной системы используют буквенные оценки, при этом A (наивысший балл) соответствует 4, B — 3, C — 2, D — 1, F — 0. Считается, что если GPA выше 3, то об этом стоит упомянуть в резюме. В почетное общество Phi Theta Kappa (<http://www.ptk.org/>) входят студенты, у которых GPA не меньше 3,5. — Примеч. перев.

² K&R (Kernighan and Ritchie) — The C Programming Language (Б. Керниган, Д. Риччи. Язык программирования C) — классический учебник по программированию на языке C, написанный его разработчиками. — Примеч. перев.

³ CLRS (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein) — Introduction to Algorithms (Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ) — популярнейшая книга по алгоритмам и структурам данных.

Собеседование — не университетский экзамен, там вас ждут реальные вопросы практические задачи.

Книга «Карьера программиста» основана на опыте моего практического участия во множестве собеседований, проводимых лучшими компаниями. Это квинтэссенция интервью с множеством кандидатов, результат ответов на тысячи вопросов, задаваемых кандидатами и интервьюерами в ведущих мировых корпорациях. В эту книгу из тысяч задач и вопросов были отобраны 150 наиболее интересных.

Мой подход

В данной книге основное внимание уделено задачам алгоритмизации, программирования и дизайна. Почему? Ответы на «поведенческие» вопросы могут варьироваться, и ваше резюме. В большинстве фирм предпочитают задавать тривиальные вопросы (например, «Что такое виртуальная функция?»), по ответам на которые легко можно понять уровень подготовки кандидата. Я расскажу и о таких вопросах, но прежде всего я хотела бы уделить внимание более сложным вещам.

Моя страсть

Моя страсть — преподавание. Мне нравится помогать людям совершенствоваться и узнавать новое.

Мой первый «официальный» преподавательский опыт я получила в колледже Пенсильванского университета на должности ассистента преподавателя, это был курс информатики.

Как инженеру Google, мне всегда нравилось обучать и воспитывать «нуглеров» (pooglers) — так в Google называют новичков. Я уделяла 20 % времени преподаванию курса информатики в Вашингтонском университете.

Эта книга и сайт CareerCup.com — отражения моей страсти к преподаванию. Даже сейчас вы можете найти меня на CareerCup.com, где я помогаю пользователям.

Присоединяйтесь к нам!

Эйл Лакман Макдауэлл

Зарегистрируйтесь на сайте <http://www.crackingthecodinginterview.com>, чтобы получить доступ ко всем решениям, обновлениям, принять участие в обсуждении задач, приведенных в этой книге, и разместить свои резюме.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Часть I Процесс собеседования

Небольшое вступление

Способы проведения собеседования у всех компаний практически одинаковы. Мы вкратце рассмотрим, как компании проводят собеседование и какова его цель. Эта информация поможет вам подготовиться и покажет, как правильно себя вести на собеседовании.

Если ваше резюме покажется работодателю интересным, вас ожидает так называемое предварительное (скрининговое) собеседование. Обычно оно проводится по телефону.

Но пусть «удаленная» форма этого собеседования не введет вас в заблуждение. Вам могут быть заданы вопросы по кодингу (программированию) и алгоритмизации, а от полученных результатов будет зависеть ваша дальнейшая судьба. Если вы хотите заранее знать специфику собеседования, узнайте должность вашего интервьюера. Инженеры и специалисты любят задавать технические вопросы.

Некоторые компании для обмена информацией при собеседовании используют онлайн-редакторы, но будьте готовы к тому, что вас попросят записать код на бумаге и продиктовать его по телефону. В некоторых случаях вы можете получить «домашнее задание», а написанный код потребуется отправить по электронной почте.

После одного или двух успешных предварительных интервью вам предложат пройти очное собеседование.

Основное собеседование состоит из 4–6 небольших интервью. Одно из них обычно носит неформальный характер, и на нем будут заданы вопросы о ваших интересах и вашем отношении к корпоративной культуре. Остальные — технические интервью — состоят из множества вопросов по алгоритмизации и программированию. Кроме того, будьте готовы ответить на вопросы по вашему резюме.

Затем интервьюеры встречаются, чтобы обсудить результаты и принять решение. Обычно это решение доводится до соискателя в течение недели.

Если за этот срок с вами никто не связался, проявите инициативу. Молчание еще не означает, что вам отказали. С вами обязательно свяжутся, когда будет принято окончательное решение.

Задержка с ответом — это не редкость. Свяжитесь с вашим нанимателем, если прошло много времени, но будьте вежливы. Наниматели такие же люди, как и вы, — они бывают заняты и им свойственна забывчивость.

Как выбираются вопросы

Кандидаты часто спрашивают меня, какие вопросы задавались на последнем собеседовании в той или иной компании, наивно полагая, что вопросы не меняются. Но то, какие вопросы будут заданы, зависит только от интервьюера.

В крупных компаниях интервьюеры проходят подготовку на специальных курсах. Так, во время своей работы в Google я посещала курсы интервьюеров. Половина

учебного времени была посвящена морально-этическим вопросам собеседования — никогда не спрашивайте кандидата о его семейном положении, национальности и т. д. Вторая половина отводилась работе с «проблемными» кандидатами. Дело в том, что некоторые люди неадекватно реагируют на вопросы, касающиеся программного кода, считая что интервьюер сомневается в их профессионализме.

По окончании курсов мне была предоставлена возможность поприсутствовать на паре реальных собеседований и прочувствовать, как все происходит в действительности. Только после этого я считалась достаточно подготовленной, чтобы проводить самостоятельные собеседования.

Это всё, чему нас учили на этих курсах, и для большинства компаний такое обучение типично. На курсах не рассматривалась тема «Обязательные вопросы собеседования Google», никто не оглашал список типичных вопросов и не требовал избегать каких-либо профессиональных тем.

Так откуда же берутся вопросы? Да откуда угодно!

Некоторые интервьюеры используют вопросы, заданные им самим, когда они были кандидатами. Некоторые обмениваются вопросами с коллегами. Некоторые ищут вопросы в Интернете и даже бессовестно используют вопросы, взятые с сайта CareerCup.com. А отдельные интервьюеры используют творческий подход: они находят готовые вопросы любым из перечисленных выше способов, а потом видоизменяют их.

Компании чаще всего не предоставляют интервьюерам списки вопросов. Интервьюер приходит с собственным списком, который содержит как минимум пять вопросов.

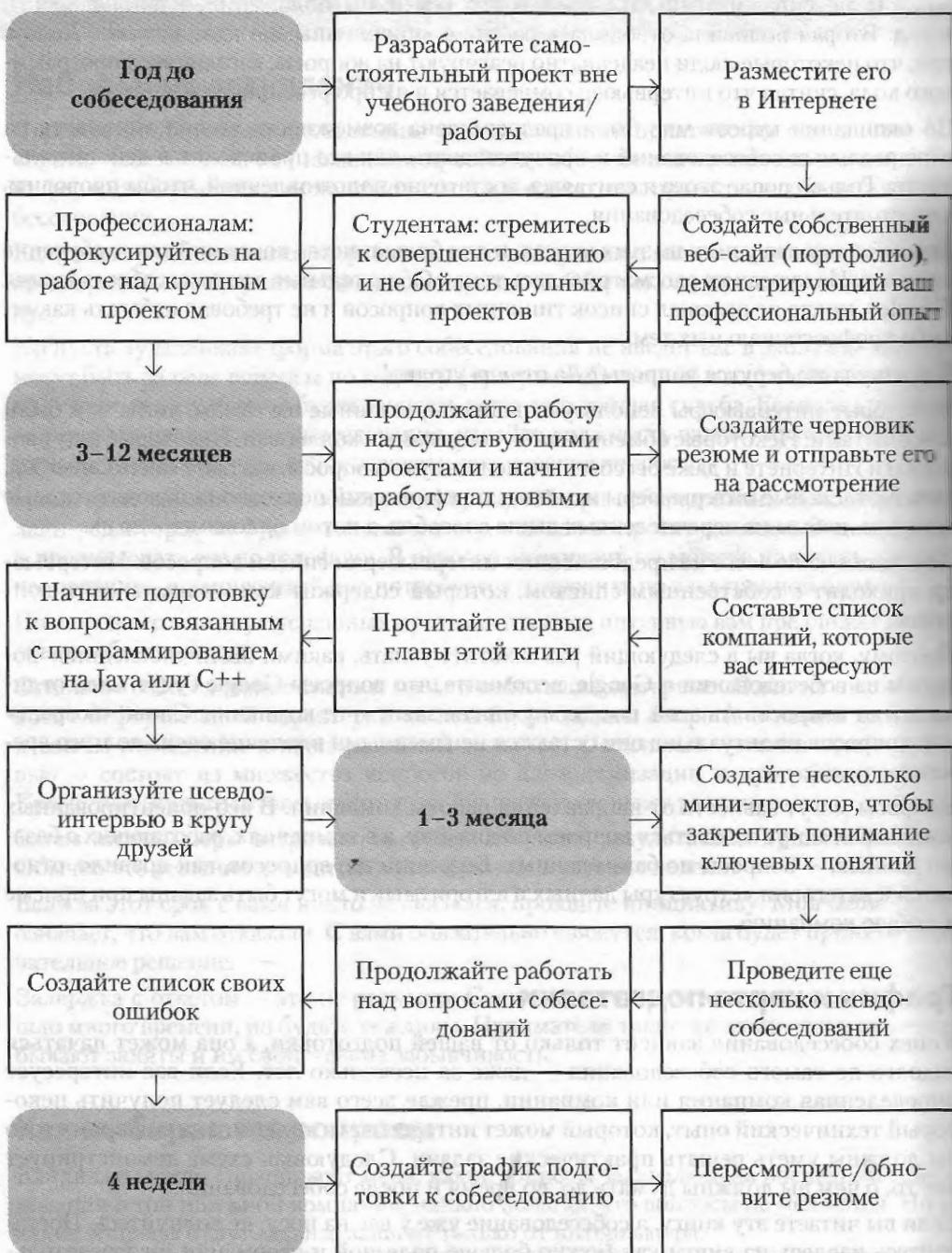
Поэтому, когда вы в следующий раз захотите узнать, какими были «последние» вопросы на собеседовании в Google, вспомните, что вопросы Google будут мало отличаться от вопросов Amazon, поскольку они не зависят от компании. Слово «возраст» для вопросов не актуально: они остаются неизменными в течение очень долгого времени.

Вопросы могут зависеть от направления работы компании. В веб-ориентированных компаниях могут задаваться вопросы по дизайну, а в компаниях, работающих с базами данных, — вопросы по базам данных. Большинство вопросов, как правило, относятся к категории «структуры данных и алгоритмы» и могут быть заданы при приеме в любую компанию.

График и карта подготовки

Успех собеседования зависит только от вашей подготовки, а она может начаться задолго до самого собеседования — даже за несколько лет. Если вас интересует определенная компания или компании, прежде всего вам следует получить некоторый технический опыт, который может интересовать эту компанию, а кроме того вы должны уметь решать практические задачи. Следующая схема демонстрирует все то, о чем вы должны думать до, во время и после собеседования.

Если вы читаете эту книгу, а собеседование уже у вас на носу, не волнуйтесь. Постарайтесь извлечь из книги как можно больше полезной информации и сосредоточьтесь на подготовке к собеседованию. Удачи!





Процедура оценки

На вопрос, как оцениваются кандидаты, большинство HR-менеджеров ответят, что используются четыре критерия: предшествующий опыт, корпоративная культура, навыки программирования и аналитические способности. Эти четыре кита без сомнения важны, но на деле все сводится к оценке ваших навыков программирования и аналитических способностей (интеллекта). Именно поэтому большая часть этой книги посвящена улучшению ваших навыков программирования и алгоритмизации. Но это не означает, что вы можете игнорировать первые два критерия.

В крупных ИТ-корпорациях ваш предшествующий опыт может не играть решающей роли, но оказать влияние на дальнейшее собеседование. Например, если вы упомянете о ранее написанной программе, а интервьюер решит, что она превосходна, то, в конечном счете, он может простить вам некоторые ошибки. Ведь собеседование – это не экзамен. Так что стоит потратить немного времени и вспомнить свой успешный программистский опыт.

Корпоративная культура (сюда же относится и ваше отношение к компании) важна скорее для небольших компаний, а не для крупных. Она определяет, будете ли вы принимать самостоятельные решения или над вами постоянно должен стоять руководитель.

Вот еще на что нужно обратить внимание:

- Если интервьюер посчитает, что вы высокомерны или склонны, вас, несмотря на всю вашу гениальность, скорее всего, не возьмут на работу. Даже суперзвезде можно отказать, если окажется, что она не способна работать в коллективе.
- Подготовьтесь к вопросам о вашем резюме. Это хоть и не решающий фактор, но довольно значимый. Потратьте толику времени на прочитывание вашего резюме, это позволит вам не сесть в лужу на собеседовании.
- Сфокусируйтесь на главном – на вопросах алгоритмизации и кодирования.

Всегда помните, что интервьюирование – это не точная наука. Решение принимается под влиянием множества случайных факторов. Любая группа людей, проводящих собеседование, субъективна: вы можете больше понравиться как человек, нежели как специалист. По отношению к вашему профессионализму это будет не очень справедливо, но тем не менее поможет пройти собеседование.

И наконец: с отказом жизнь не заканчивается. В течение года вы можете повторно попытаться пройти собеседование: многие кандидаты, изначально получившие отказ, позднее получали приглашение на работу.

Неправильные ответы

Одним из распространенных и опасных заблуждений является то, что кандидат должен абсолютно правильно и грамотно ответить на все поставленные вопросы. Это утверждение далеко от истины.

Во-первых, ответы на вопросы собеседования не оцениваются по принципу «правильный–неправильный». Когда я даю оценку кандидату, то никогда не подсчитываю количество правильных ответов на вопросы. Скорее я оцениваю оптимальность вашего решения, сколько времени было потрачено и насколько чист ваш код. Это не двоичная оценка – «да» (1) или «нет» (0), на результат оказывает влияние множество факторов.

Во-вторых, ваши результаты рассматриваются в сравнении с другими кандидатами. Вы нашли оптимальное решение за 15 минут, а кто-то решил ту же задачу за пять. Означает ли это, что он лучше вас? Может быть, да, а может и нет. Если вам задаются простые вопросы, то ожидается, что оптимальное решение будет получено быстро. Но если вопросы сложные, то интервьюер предполагает, что вы можете допустить ряд ошибок.

На собеседованиях в Google я видела единственного кандидата (из нескольких тысяч), который не допустил ни одной ошибки. Все остальные, включая сотни тех, кто был принят, их допускали.

Дресс-код

Разработчики программного обеспечения, как и свободные художники, одеваются достаточно неформально. Но собеседование — это как раз то мероприятие, на котором вас встретят по одежке. Существует негласное правило: ваша одежда должнаividno отличаться от одежды других кандидатов на эту же вакансию.

Я даже могу дать вам некоторые рекомендации при выборе одежды для собеседования. Знание правил дресс-кода обеспечит вам попадание в «безопасную зону» — нужно выглядеть не слишком модным, но и не слишком обычным. Многие люди, приходящие на собеседование в крупную компанию в драных джинсах и футболках, получили работу. В конце концов, интервьюеры в первую очередь оценивают навыки программирования, а не внешний вид.

	Стартап	Microsoft, Google, Amazon, Facebook и другие крупные компании	Банки
Мужчина	Хаки, слаксы (брюки) или приличные джинсы. Рубашка с коротким рукавом или обычная костюмная рубашка	Хаки, слаксы или приличные джинсы. Рубашка с коротким рукавом или обычная костюмная рубашка	Костюм, без вариантов
Женщина	Слаксы или приличные джинсы. Приличные блузка или свитер	Слаксы или приличные джинсы. Приличные блузка или свитер	Костюм или брюки с приличной блузкой

Все это — абстрактные рассуждения, вы должны учитывать специфику компании, где будете проходить собеседование. Если вы планируете получить должность проект-менеджера, ведущего разработчика или любую другую административную должность, то вы должны одеться более строго.

10 наиболее частых ошибок

1. Использование компьютера

Стремите ли вы учиться серфингу в бассейне? Скорее всего, нет. Ведь вам нужны волны — другие особенности «живой природы». Могу спорить, что ваш выбор падет на канарское побережье.

Использование компилятора для «репетиции» собеседования подобно тренировкам в бассейне. Забудьте про компилятор, возьмите ручку и лист бумаги. Используйте компилятор для проверки решения, но только после того, как написали и протестирували код.

2. Игнорирование поведенческих вопросов

Многие кандидаты тратят все свое время на подготовку к техническим вопросам и упускают поведенческие. Но ваш интервьюер, скорее всего, их не упустит!

Более того, ответы на поведенческие вопросы могут изменить восприятие интервьюером ваших профессиональных данных. К ответам на поведенческие вопросы легко подготовиться. Вспомните все свои проекты и используйте их для подготовки.

3. Отказ от псевдоинтервью

Представьте, что вы готовитесь к публичному выступлению перед своей группой, компанией или какой-нибудь другой большой аудиторией. Ваше будущее зависит от этого. Только сумасшедшие или излишне самоуверенные люди будут готовить такую речь в одиночку.

Один из способов подготовки — так называемое псевдоинтервью. Если вы инженер, то должны быть знакомы с коллегами. Попросите приятеля провести для вас «собеседование». Этот метод принесет только пользу!

4. Попытка зазубрить ответ

Запоминание решения конкретной задачи пригодится вам только в том случае, если интервьюер даст именно эту задачу, но никак не поможет решить новые задачи. Очень маловероятно, что в этой книге вы найдете все или хотя бы большую часть вопросов, которые могут достаться вам на собеседовании.

Намного эффективнее не привязываться к конкретике. Это поможет вам разработать стратегии решения новых задач. Лучше качество, нежели количество.

5. Решение задачи «в уме»

Открою вам секрет — я не телепат и не знаю, что происходит у вас в голове. Следовательно, если вы ничего не произносите вслух, не известно, о чем вы думаете. Если вы молчите, я считаю, что у вас нет решения. Больше говорите и пытайтесь комментировать решение. Это покажет интервьюеру, что вы решаете поставленную задачу, а ваше молчание будет расценено как то, что вам нечего сказать.

Дайте возможность интервьюеру подсказать вам путь решения или указать на ошибку, когда вы в этом будете нуждаться. Простейшие навыки коммуникации помогут получить желаемую вакансию. Что может быть лучше?

6. Спешка

Программирование — это не ралли, даже на собеседовании. Уделите больше времени написанию кода. Спешка приводит к ошибкам, а они могут быть расценены как небрежность. Пишите код последовательно и неторопливо, анализируйте задачу и проверяйте код. В итоге вы закончите работу за меньшее время и с минимальным количеством ошибок (а может и без них).

7. Грязный код

Знаете ли вы, что код, даже написанный без ошибок, может быть просто ужасным? К сожалению, это так! Дублирование, огромные структуры данных (отказ от объектно-ориентированного программирования) и т. д. являются показателями плохой программы! Когда вы пишете код, представьте, что он должен быть «ремонтопригодным». Разбейте код на подпрограммы и выберите оптимальную структуру, соответствующую данным.

8. Отказ от проверки

Когда вы пишете код в реальной жизни, вы его тестируете, так почему бы этого не сделать на собеседовании? Когда код написан, «запустите» его (в качестве компилятора будете выступать вы сами) и протестируйте. При решении сложных задач тестируйте фрагменты кода по мере написания.

9. Небрежное отношение к исправлению ошибок

Ошибка неизбежны. Это норма жизни и программирования. Если вы тщательно протестируете код, то наверняка обнаружите свои ошибки, — и это хорошо.

Если вы обнаружили ошибку, то прежде чем исправить ее, постарайтесь разобраться, откуда она появилась. Некоторые кандидаты, обнаруживая, что при определенных условиях функция возвращает `false`, просто инвертируют значение, а не разбираются, почему так происходит. Конечно, это встречается не часто, но подобное действие приводит к дополнительным ошибкам и показывает, насколько вы небрежно отноитесь к написанию кода.

Никто не застрахован от ошибок, но бездумное исправление кода недопустимо.

10. Отказ от решения

Очень часто вопросы оказываются достаточно сложными. Примете ли вы вызов или повернете назад? Я рекомендую достойно встретить трудную задачу. Ведь собеседования для этого и проводятся. Не удивляйтесь, когда вам достанется действительно сложный вопрос.

Часто задаваемые вопросы

Нужно ли мне говорить интервьюеру, что я уже знаком с вопросом?

Вы обязательно должны сказать своему интервьюеру, что знаете вопрос (и ответ на него). Конечно, многим людям это может показаться глупым, ведь если вы знаете вопрос и ответ, то с легкостью справитесь, правильно? Не совсем так.

Существует несколько причин, по которым вы должны сказать экзаменатору, что знакомы с этим вопросом:

- Вы завоюете расположение интервьюера. Вы продемонстрируете свою честность, а это много значит. Помните, что интервьюер оценивает вас как потенциального товарища по команде, и в этом случае вы проявите себя как честный и порядочный человек.
- Постановка задачи могла быть изменена. Не нужно рисковать, повторяя неправильный ответ.

- Если вы легко и быстро дадите правильный ответ, интервьюер сам догадается, что вы знакомы с этой задачей. Он знает уровень сложности задания, и быстрое (и правильное) решение будет выглядеть подозрительным.

Какой язык программирования следует использовать?

Многие говорят, что можно использовать любой язык, лишь бы вы им отлично владели, но в идеале это должен быть язык, которым пользуется интервьюер. Обычно я рекомендую писать код на C, C++ или Java, поскольку большинство интервьюеров владеют этими языками. Мое личное предпочтение — Java (за исключением задач, требующих C/C++), поскольку этот язык прост и понятен любому человеку, даже если он программирует на C++. Именно по этой причине решения всех задач в этой книге выполнены на Java.

После собеседования мне ничего не сказали. Мне отказали?

Нет. Практически все компании сообщают своим кандидатам об отказе. Если вам сразу ничего не сказали, то это ничего не означает. Возможно, вы приняты, но у вашего агента (представителя рекрутинговой компании) выходной, и он сообщит вам об этом позже. В компании может производиться реорганизация, и еще не ясно, сколько сотрудников понадобится. Хотя может оказаться, что вы действительно не подходите, но рекрутинговая компания не удосужилась сообщить вам об этом. Довольно странно для компании отказать кандидату и не сообщить о своем решении. В интересах нанимателя уведомить вас об окончательном решении, как только оно принято.

Могу ли я попытаться еще раз, если мне отказали?

Практически всегда можно сделать еще одну попытку, но нужно сделать паузу (обычно от 6 месяцев до одного года). Плохие результаты наверняка окажут негативное влияние на результаты повторного собеседования. Но многие люди, которым отказал Google или Microsoft, впоследствии получили работу.

5. Решение задачи «в уме»

Открою вам секрет: в 99% случаев в ходе собеседования вы можете решить задачу «в уме». Для этого вам нужно не пытаться решить ее в уме, а просто напомнить себе, что вы можете это сделать. Используйте технику, описанную в главе 3, чтобы вернуться в свое прошлое и вспомнить, как вы решали задачи в школе. Если вы в этом будете уверенны, то у вас не возникнет проблем с решением задачи в уме. А если вы не уверены, то лучше всего решить задачу на бумаге, а потом переписать ее в уме.

На бывшем месте этого пункта я написал: «Возвращайтесь в свое прошлое, чтобы решить задачу». Но это не совсем точно, потому что в ходе собеседования вы должны решить задачу, не вспоминая прошлое. Вам нужно вернуться в свое прошлое, чтобы вспомнить, как решали задачи в школе. А затем вернуться в настоящий момент и решить задачу, не вспоминая прошлое. Итак, вы должны вернуться в свое прошлое, чтобы вспомнить, как решали задачи в школе, а затем вернуться в настоящий момент и решить задачу, не вспоминая прошлое.

Часть II За кулисами

Для большинства кандидатов собеседование похоже на черный ящик. Вы входите, получаете вопросы и выходите... с предложением работы или без него.

Задавались ли вы вопросами:

- Как принимаются решения?
- Общаются ли интервьюеры друг с другом?
- Что интересует компанию?

Скоро вы узнаете ответы!

Специально для этой книги мы нашли экспертов-интервьюеров от пяти ведущих компаний — Microsoft, Google, Amazon, Yahoo! и Apple, — чтобы узнать из первых рук, что происходит там, за кулисами.

Эти эксперты рассказали нам про типичное собеседование и поведали, что происходит после того, как вы уходите.

От них мы узнали, чем различаются собеседования — от Amazon до Google. У каждой компании есть особенности, знание которых позволит вам избежать ситуации, когда один строгий экзаменатор из Amazon или два из Apple указывают вам на дверь.

Кроме того, эксперты поведали нам, на что в первую очередь обращают внимание. Компании, занимающиеся разработкой программного обеспечения, фокусируются на алгоритмах и коде. В других компаниях могут быть другие специфические приоритеты. Зная особенности компании, вы сможете лучше подготовиться.

Поэтому присоединяйтесь к нам, поскольку мы собираемся заглянуть за кулисы Microsoft, Google, Amazon, Yahoo! и Apple.

Microsoft

Microsoft ищет умных людей — фанатов, увлеченных технологиями. Скорее всего, здесь не потребуют от вас идеального знания 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 адвокатом в том случае, если вы провалитесь на первом собеседовании. Он поможет вам попасть на повторное собеседование. Рекрутеры могут отстаивать ваши интересы вне зависимости от того, будете вы наняты или нет.

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 прочувствовать командный дух.

Интервьюеры могут сказать, какое впечатление вы на них произвели (или, наоборот, не сказать).

Когда интервью 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 от рекрутера, напомните ему о себе по электронной почте.

Не забывайте, что отсутствие ответа может означать лишь то, что ваш рекрутер расцененный или очень занятой человек, а не то, что вам отказали.

ПОДГОТОВЬСЯ!

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 превосходный программный продукт. Например, я использовал Visual Studio для программирования игр, а API просто превосходен». Покажите свое увлечение технологией!

ОСОБЕННОСТИ

Вы будете иметь беседу с менеджером по кадрам только в том случае, если вы прошли собеседование. Расценивайте разговор с менеджером как хороший знак.

Amazon

В Amazon процесс обычно начинается с пары скрининговых 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 в несколько команд. Реже ограничиваются одним скрининговым интервью. Так происходит, если кандидат знаком интервьюерам или недавно проходил собеседование на другую вакансию.

Инженер, проводящий собеседование, обычно просит написать небольшой фрагмент кода, используя специальный редактор, например CollabEdit¹. Зачастую интервьюер интересует, какими технологиями вы владеете.

Затем вас приглашают в Сиэтл. 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.

У собеседования в Amazon есть особенность — главный интервьюер (*bar riser*). Он проходит специальную подготовку и может общаться с кандидатами вне пределов их группы, чтобы сбалансировать группу в целом. Если вопросы одного из интервьюе-

¹ Данный редактор используется для совместной работы над документами. То есть интервьюер будет видеть все изменения, вносимые вами в документ, в реальном времени. — Примеч. перев.

ров оказываются более сложными, чем у остальных, скорее всего, это главный интервьюер. У него огромный опыт в проведении собеседований, и его решение может быть окончательным. Помните: если в собеседовании с главным интервьюером вы показали себя с хорошей стороны, это еще не означает, что решения других интервьюеров не будут учитываться.

Как только интервьюеры выставили оценки, они встречаются, чтобы принять окончательное решение.

В большинстве случаев HR-менеджеры от Amazon превосходно выполняют свои обязанности, но и у них бывают накладки. Если вы не получили ответ в течение недели, напомните о себе по электронной почте.

ПОДГОТОВИТЬСЯ!

Amazon — веб-ориентированная компания, и она беспокоится о масштабировании. Убедитесь, что вы готовы к вопросам о модульном наращивании систем. Для лучшей подготовки к вопросам интервьюеров прочитайте главу «Масштабируемость и ограничения памяти». В Amazon любят задавать вопросы по объектно-ориентированному программированию. Прочтите главу 8 «Объектно-ориентированное проектирование».

ОСОБЕННОСТИ

Вы должны произвести должное впечатление на главного интервьюера и менеджера по найму, если хотите пройти собеседование и получить должность.

Google

Ходят много страшных рассказов о прохождении собеседования в Google, но это только слухи. На самом деле собеседование в Google не очень отличается от собеседования в Microsoft или Amazon.

Прежде всего инженер Google побеседует с вами по телефону, так что ожидайте технических вопросов. Эти вопросы могут включать в себя написание кода, иногда через тестную работу с документом. Обычно кандидатам задаются одинаковые вопросы при телефонном собеседовании, и на личной встрече.

Личном собеседовании с вами будут общаться от четырех до шести интервьюеров, числе и главный интервьюер (lunch interviewer). Решение интервьюера является тайной, никто не знает, что думает его коллега. Вы можете быть уверены, что интервьюер дает независимую оценку.

Интервьюеров нет согласованной «структурой» или «системы» вопросов. Вам могут задать любой вопрос, который посчитают нужным.

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, коммуникативные способности), по каждому из них выставляются оценки от 1.0 до 4.0. Интервьюеры обычно не участвуют в работе комитета по подбору персонала, поэтому они не могут повлиять на решение.

Важнейшую роль играют ваши оценки. Комитет хочет видеть ваши преимущества по отношению к другим кандидатам. Оценки 3.6, 3.1, 3.1, 2.6 предпочтительнее, чем 3.1.

Не обязательно становиться лучшим на каждом собеседовании, телефонное собеседование тоже не является решающим фактором.

Если комитет по подбору персонала принял 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, заинтересованная в разработке наращиваемых модульных систем. Поэтому убедитесь, что вы изучили вопросы из главы «Масштабируемость и ограничения памяти». Интервьюеры Google любят задавать вопросы из раздела «Поразрядная обработка», поэтому убедитесь, что вы разобрались в этой теме.

ОСОБЕННОСТИ

Ваши интервьюеры не принимают решение о найме, их оценки передаются в комитет по подбору персонала, который дает рекомендации (принять или отказать) исполнительному комитету Google.

Apple

Apple — наименее бюрократичная компания. Интервьюеры будут оценивать не только технические навыки, для них очень важно ваше отношение к компании. Желательно, чтобы соискатель был пользователем Мака или, по крайней мере, был знаком с этой системой.

Интервью обычно начинается с телефонного звонка рекрутера — он должен оценить ваши базовые навыки, затем происходит серия технических интервью с членами команды.

После приглашения в кампус рекрутер вкратце расскажет вам о процедуре собеседования. Затем вас ждет 6–8 собеседований с членами команды. Это позволит вам познакомиться с будущими коллегами.

Вам предстоит смесь собеседований 1-на-1 и 2-на-1. Будьте готовы писать программный код и убедитесь, что умеете четко формулировать свои мысли. Обед с возможным будущим начальником может показаться случайностью, но все это — продолжение собеседования. Каждый интервьюер занимается собственной областью и обычно не обсуждает ваши результаты с другими.

ПОДГОТОВИТЬСЯ!

Если вы знаете, какая команда будет проводить собеседование, убедитесь, что знакомы с продуктом этой команды. Нравится ли он вам? Что вы хотели бы улучшить? Продемонстрируйте свою заинтересованность.

ОСОБЕННОСТИ

Интервью часто проводятся в режиме 2-на-1, не беспокойтесь, он не отличается от режима 1-на-1. Не забывайте, что сотрудники Apple должны быть фанатами своей продукции, поэтому продемонстрируйте заинтересованность.

В конце дня интервьюеры обмениваются мнениями. Если они чувствуют, что вы — потенциальный кандидат, то дальше вас ждет интервью с директором и вице-президентом компании. Хотя такое собеседование еще ничего не гарантирует, но это хороший знак. Решение принимается за кулисами, и даже если вы не подходите, то просто покинете здание, думая, что вы самый лучший.

После окончания собеседования с директором и вице-президентом интервьюеры уединяются в конференц-зале, чтобы принять окончательное решение. Обычно вице-президент не присутствует на таком совещании, но у него есть право вето, если вы не произвели на него должного впечатления. Через несколько дней вы узнаете об окончательном решении от рекрутера, но не стесняйтесь спросить его пораньше.

Facebook

Хотя при решении онлайновых инженерных головоломок существует много способов сжульничать, это еще один способ обратить на себя внимание. Просто продолжайте решать задачки, не забывая подать традиционное заявление о приеме на работу через онлайн-службу или ярмарку вакансий.

Перед тем как вас пригласят на очное собеседование, вы пройдете как минимум два телефонных интервью; они будут носить технический характер, и вам придется написать программный код в Etherpad или аналогичном онлайн-редакторе.

Чаще всего интервью проводят разработчики программного обеспечения, но в нем могут принимать участие и менеджеры по подбору персонала.

Каждому интервьюеру отводится строго определенная роль, это позволяет гарантировать, что вопросы не будут дублироваться, и в итоге представление о кандидате получится более разносторонним. Вопросы разбиваются на группы: алгоритмы, навыки программирования, архитектура/навыки проектирования, кроме того, оцениваются ваши возможности оперативно реагировать на быстро изменяющуюся среду Facebook.

Интервьюеры пишут отзывы и обсуждают ваши достижения только после окончания собеседования. Это гарантирует, что успех (или, наоборот, неудача) на одном из собеседований никак не повлияет на результаты следующего.

Как только отзыв составлен, интересующаяся в вас команда и менеджер по подбору персонала собираются, чтобы принять окончательное решение. 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 ИТ-компаний нужны инициативные разработчики. Покажите, что вы инициативны и можете быстро работать.

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 — улучшить ваши навыки программирования. Вы получите задания от старших разработчиков, изучите новые методы и, в конечном счете, подниметесь на уровень выше, чем были до собеседования.

Yahoo!

Yahoo! набирает претендентов из двадцати ведущих учебных заведений, осталые кандидаты могут попытаться счастья через доску вакансий Yahoo! (или, что еще лучше, по знакомству). Если вас отобрали на собеседование, оно начнется с телефонного звонка. С вами будет беседовать ведущий разработчик или менеджер.

На протяжении собеседования вам предстоит пройти 6–7 интервью длительностью около 45 минут каждое. Интервьюеры специализируются на конкретных областях. Например, один экзаменатор специализируется на базах данных, другой — на компьютерной архитектуре и т. д. Обычно распорядок интервью следующий:

- 5 минут — общее знакомство, вы рассказываете о себе, о своих проектах и т. д.
- 20 минут — вопросы по кодингу. Вас могут попросить, например, реализовать сортировку слиянием.
- 20 минут — системное проектирование. Вам предложат спроектировать большой распределенный кэш. Эти вопросы часто строятся на вашем предыдущем опыте.

В конце дня вы, скорее всего, встретитесь с ведущим разработчиком (руководителем). Разговор может быть о чем угодно. Вы можете просматривать демоверсии продуктов, рассуждать о развитии компании или вашей роли в ней. Обычно такая беседа не является решающим фактором.

Тем временем интервьюеры обсудят ваши результаты и попытаются прийти к решению. Окончательное решение принимает менеджер по подбору персонала, взглянув на ваши сильные и слабые стороны.

Если вы подходите, то вы чаще всего (но не всегда) узнаете об этом в тот же день. Существует множество причин, по которым на принятие решения понадобится несколько дней. Например, есть еще несколько кандидатов, которые должны пройти собеседование.

ПОДГОТОВИТЬСЯ!

Как правило, в Yahoo! задают вопросы, связанные с системным проектированием. Поэтому подготовьтесь к ним. Вы должны уметь не только написать код, но и разработать программное обеспечение.

ОСОБЕННОСТИ

Интервью проводится не рядовыми менеджерами по набору персонала. Для Yahoo! характерно то, что решение (приняты вы или нет) обычно принимается в день собеседования. Ваши интервьюеры обсуждают результаты, пока вы встречаетесь с последним экзаменатором.

Часть III Нестандартные случаи

Кандидат-профессионал

Если вы внимательно прочитали предыдущую главу, то следующая информация вас не удивит: кандидатам, имеющим профессиональный опыт, задаются те же вопросы, что и их неопытным коллегам, — подход меняется незначительно.

Большинство вопросов, как вы уже знаете, посвящены структурам данных и алгоритмам. Крупные компании считают, что это хорошая проверка ваших возможностей.

Некоторые интервьюеры оценивают опытных кандидатов более строго. Если у человека большой опыт работы, он должен лучше справиться с заданием, не правда ли?

Однако другие интервьюеры имеют противоположное мнение. Опытные кандидаты давно окончили университет, поэтому успели многое забыть, таким образом, их нужно оценивать по более низкому стандарту.

В среднем же подходы различных интервьюеров более менее сбалансированы. Так что, даже если вы опытный кандидат, вам, скорее всего, будут предложены те же вопросы, что и прочим.

Исключением из этого правила являются вопросы по системному дизайну и архитектуре, а также вопросы по резюме.

Обычно студенты не любят заниматься системной архитектурой, поэтому знания они получают только во время работы. Следовательно, ваш результат на собеседовании по этой теме прямо пропорционален вашему опыту. Однако эти же вопросы задают и более опытным кандидатам, поэтому вы должны быть готовы решить их так, как можете.

Интервьюеры ожидают, что опыт кандидата позволит ему дать более развернутый ответ на поставленный вопрос. Ответ более опытного претендента на вопрос «Какая ошибка, допущенная вами, была самой серьезной?» будет более интересным и глубоким.

Тестеры и SDET

О вакансии SDET (Software Design Engineer in Test, программист-тестер) следует поговорить отдельно. Специалисты в этой области должны быть не только прекрасными программистами, но и прекрасными тестерами.

Если вы претендуете на должность тест-программиста:

- Подготовьтесь к базовым задачам тестирования. Как вы будете тестировать лампочку? Ручку? Кэш-регистр? Microsoft Word? Глава 12 «Тестирование» поможет ответить на эти вопросы.
- Подготовьтесь к заданиям, связанным с программированием, — тестер обязан уметь программировать. Хотя требования к SDET ниже, чем для SDE (Software Design Engineer, разработчик программного обеспечения), вам необходимо разбираться в алгоритмах. Убедитесь, что вы можете справиться с заданиями по алгоритмизации и программированию, которые даются обычным разработчикам программного обеспечения.

- Совершенствуйте навыки тестера и программиста. В качестве примера возьмем очень популярное задание: «Какой код нужно написать, чтобы получить X?» Далее обычно следует: «Ок, а теперь протестируйте его». Даже если вас об этом не спросили, уточните: «Как именно нужно протестировать код?» Помните: любое задание может превратиться в тест-задачу.

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

Совет

В завершении этого раздела хочу дать вам небольшой совет, способствующий карьерному росту. Если вы, как и многие другие кандидаты, расцениваете должность тестера как стартовую должность в компании, не обольщайтесь. Для многих кандидатов переход с этой должности на должность разработчика программного обеспечения оказался трудным испытанием. Если вы все-таки решились на этот шаг, убедитесь в том, что сохранили все свои навыки программирования и алгоритмизации, и попробуйте осуществить переход в течение одного-двух лет — чем больше пройдет времени, тем сложнее это сделать.

Не позволяйте атрофироваться навыкам программирования.

Менеджеры программ и менеджеры продукта

PM¹ — общая аббревиатура, которой обозначают как менеджеров программ, так и менеджеров продукта. Но роли и задачи этих двух РМ сильно различаются даже в пределах одной компании. В Microsoft, например, некоторые РМ, по сути, выполняют функции маркетологов, то есть больше общаются с клиентами, нежели программируют. А вот в других компаниях РМ могут провести большую часть своего рабочего дня, занимаясь программированием.

Когда интервьюеры ищут кандидата на должность РМ, 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. Поэтому постарайтесь продемонстрировать, что вы не остановитесь, займетесь поиском новой информации, расстановкой приоритетов и решите задачу. Обычно кандидатов не просят решать конкретную задачу (хотя и такое может быть), достаточно рассказать, что вы будете делать в такой необычной ситуации.
- *Потребительский фокус.* Интервьюеры хотят видеть, что вам хорошо знакома целевая аудитория. Вы твердо уверены, что все потенциальный 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 буде-

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. перев.

пользоваться программой? Ожидайте заданий типа «Разработайте будильник для слепого». Вы должны понимать, кто является вашей целевой аудиторией и как он будет использовать продукт. Необходимые навыки описаны в главе 12 «Тестирование».

- *Потребительский фокус (технические навыки)*. Некоторые команды, работающие со сложными продуктами, хотят убедиться, что их РМ понимают сам продукт. Трудно быть эффективным менеджером продукта или программы, если не знаешь, как это работает. Близкое знакомство со всеми существующими клиентами мгновенного обмена сообщениями, вероятно, окажется бесполезным для работы в команде MSN Messenger, но понимание задач безопасности является необходимым условием для работы в Windows Security.
- *Коммуникативные способности*. РМ должен уметь общаться с сотрудниками компаний 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 легко оцениваются 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 прозвучать, что вы увлекаетесь новыми технологиями и можете работать в команде. Вас могут спросить: «Чем вам привлекает Microsoft?» Интервьюеры ожидают увидеть энтузиазм в вашем рассказе о предшествующем опыте и задачах команды. Они хотят видеть, что вы стремитесь решать сложные задачи.
- *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. Интервьюер будет рад видеть, что вы способны справиться с конфликтами, берете на себя инициативу, вы понимаете коллектив, и людям нравится работать с вами. Вам следует уделить должное внимание поведенческим вопросам.

Все перечисленные навыки важны для РМ и являются ключевыми для собеседования. Значимость каждой области примерно одинакова.

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. Например, требования Google к менеджерам существенно выше, чем к обычным программистам.

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. Интервьюер сразу увидит, что вы высокомерны или, наоборот, слишком мягки, чтобы стать лидером (руководителем).

- **Расстановка приоритетов.** Менеджеры часто сталкиваются со щекотливыми вопросами, например как убедиться, что команда укладывается в сроки. Ваши интервьюеры хотят видеть, что вы способны правильно разложить все «по полочкам» и расставить приоритеты.
- **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 делается слепок вашей личности.
- **Доведение дела до конца.** Менеджер должен быть человеком, который всегда добивается своей цели. Вам необходимо найти баланс между подготовкой к проекту и его реализацией. Менеджер должен знать, как структурировать проект и как мотивировать людей, только так можно достичь цели.

В конечном счете, все эти критерии относятся к вашему предыдущему опыту и к вашей личности. Убедитесь, что хорошо подготовились, и проверьте таблицу подготовки к собеседованию (см. с. 40).

Стартап

В стартапах¹ процессы подачи заявления и собеседования имеют существенные различия в зависимости от конкретной фирмы. Мы не можем рассмотреть каждый стартап, но можем обсудить некоторые общие черты. Но не забывайте, что у каждого стартапа есть особенности.

Процесс подачи заявления

Стартапы не только публикуют списки вакансий, но и могут самостоятельно заниматься подбором кадров. Приглашающий не обязательно будет вашим коллегой или другом. Достаточно разместить свое резюме на соответствующем сайте, а стартапы найдутся сами.

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, или поискать более серьезную компанию.

¹ Что же такое стартап (start-up)? Английский язык довольно изворотлив, и у понятия start-up есть множество вариантов перевода: запуск, начинание, начало, старт и т. д. Все зависит от контекста. Здесь речь идет о компании начального уровня. Итак, стартап — недавно созданная компания — обладает ограниченным набором ресурсов и возможностей. — Примеч. перев.

Резюме

Стартаперы — это не только программисты. Это люди, которые способны быть предпринимателями. В вашем резюме должны быть отражены инициативность и возможность сразу взяться за работу. Они ищут людей, знакомых с языком программирования, который используется в данной компании.

Процесс собеседования

В отличие от крупных компаний, которые оценивают вашу способность к разработке программного обеспечения, стартапы в основном смотрят на ваши персональные навыки и предшествующий опыт:

- *Подгонка личности.* Установите дружеские отношения с интервьюерами — 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.* Стартаперы 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.

Часть IV Перед собеседованием

Получаем «правильный» опыт

Хотя решение о найме принимается по результатам собеседования, огромную роль играют резюме и предыдущий опыт работы — это ваш пропуск на собеседование. Вы должны постоянно накапливать технический (и нетехнический) опыт. Повышенная науки программирования, вы неизменно остаетесь в выигрыше — и неважно, кто вы — студент или профессионал.

Для студентов могут быть полезны следующие рекомендации:

- *Принимайте участие в больших проектах.* Даже если вы еще учитесь, не отказывайтесь от участия в больших проектах. Такой опыт можно будет упомянуть в резюме, а это повысит ваши шансы на собеседованиях в крупных компаниях. Чем больше проект связан с реальными задачами, тем лучше.
- *Повышайте квалификацию.* Даже на ранних стадиях обучения вы можете получить профессиональный опыт. Первокурсники (и второкурсники) могут, например, принимать участие в программах Google Summer of Code¹. Если вы не можете принять участие в подобной программе, найдите стартап и попробуйте свои силы.
- *Попробуйте начать какой-либо проект.* Собственные проекты производят впечатление на любую компанию. Такая работа не только увеличивает ваш технический опыт, но и показывает, что вы инициативны и можете достичь поставленной цели. Используйте выходные дни для разработки собственного программного обеспечения. Если у вас есть научный руководитель, можно попытаться получить грант под вашу работу.

Профессионалы тоже должны обладать правильным опытом, который позволит попасть в компанию их мечты. Вы можете работать в Google, но мечтаете о Facebook. А может, вы работаете тестером, но стремитесь попасть в более крупную компанию или хотите заниматься разработкой программ. В этом случае 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. Стремитесь к участию в крупных проектах — это дополнительный плюс для вашего резюме.
- *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 — все пойдет вам в плюс. Выполняя такие проекты, вы получаете опыт. Страйтесь использовать новые технологии, чтобы идти в ногу со временем. Обязательно упомяните эти проекты в резюме: люди, разрабатывающие программы «ради собственного удовольствия», производят хорошее впечатление на интервьюеров.

Компании хотят увидеть, что вы умны и что вы умеете программировать. Если вы можете это доказать, у вас больше шансов пройти собеседование.

¹ Google Summer of Code (GSOC) — программа компании Google, в рамках которой ежегодно проводится отбор проектов с открытым исходным кодом, в которых могут принять участие студенты. Победителям выплачиваются денежные гранты. — Примеч. ред.

Нужно заранее думать о развитии вашей карьеры — какой дорогой она должна пойти? Если вы собираетесь идти по пути менеджмента, то, даже если в настоящий момент вы ищете вакансию разработчика, стремитесь развивать качества лидера.

Налаживаем связи

Наверняка вы слышали, что многие люди получают работу «по знакомству». Скажу даже больше — некоторые устраиваются на работу через друзей своих друзей. И это просто прекрасно. Если у вас N друзей, то друзей друзей будет уже N^2 , то есть ваши шансы получить работу существенно повышаются.

Это означает, что не только вы сами, но и ваш круг знакомств оказывает влияние на ваш шанс найти работу.

Правильный круг знакомств

Правильный круг знакомств должен быть достаточно широк и в то же время он должен быть закрытым. Кажется, что эти два понятия не совместимы, поэтому сначала рассмотрим, что такое «широкий и закрытый» круг знакомств.

- **Широкий** круг знакомств означает, что все ваши знакомые и друзья имеют отношение не только к узкой области ваших интересов. Например, бухгалтер может помочь вашей карьере, поскольку в его организации (или у его знакомых) вполне может найтись место и для вас. Будьте открыты для общения с любым человеком.
- **Закрытый** — намного проще достучаться до человека, который дружит с вашим близким другом, чем через абстрактное «шапочное» знакомство. Некоторые люди даже коллекционируют визитные карточки — у них очень много контактов, но навряд ли они могут назвать этих людей своими друзьями и знакомыми. Такой подход не поможет при поиске работы. Сделайте ваши связи более глубокими.

Постарайтесь найти баланс, не нужно «собирать карточки» — этим вы ничего не добьетесь.

Как построить сильный круг знакомств

Некоторые люди утверждают, что для знакомства достаточно выйти и встретить человека. В какой-то мере это правда. Но где такая встреча должна произойти? И как вы перейдете от простого знакомства к общению?

Вам помогут несколько простых советов:

1. *Используйте Интернет.* Сайты вроде Meetup.com или социальные сети помогут вам отслеживать события, связанные с областью ваших интересов и вашими целями. Если у вас до сих пор нет своей визитной карточки — вы студент или безработный, — срочно решите эту задачу.
2. *Не бойтесь знакомиться с людьми.* Вы волнуетесь или стесняетесь? Не переживайте, от простого «здравствуйте!» хуже вам не станет. Что может произойти? Если человеку вы не понравитесь, он просто не станет с вами знакомиться, и вы его больше не увидите.
3. *Будьте открыты, говорите о своих интересах.* Если кто-то создает стартап или рассказывает что-то интересное для вас, пригласите его на чашечку кофе.

4. Отслеживайте события на LinkedIn или пригласите человека на чашку кофе, если его стартап покажется вам интересным.
5. Будьте доброжелательны — и это главное. Люди захотят помочь вам, если вы помогли им.

Помните: круг знакомств — это не только люди, с которыми вы знакомы лично. В наши дни круг знакомств можно существенно расширить благодаря Интернету — вам помогут блоги, Facebook и электронная почта. Но не забывайте — если общение происходит только онлайн, сложнее установить связь и получить желаемое.

Идеальное резюме

Рекрутеры, просматривающие резюме, обращают внимание на те же самые детали, что и интервьюеры. Они хотят понять, насколько вы умны и обладаете ли вы навыками, необходимыми для желаемой должности.

Это означает, что вам нужно сделать акцент в резюме на «правильных» данных, и это будет не ваше увлечение теннисом, путешествиями или гаданием на кофейной гуще. Подумайте, стоит ли сокращать информацию технического характера, заменяя ее описанием разнообразных хобби.

Правильный размер

Обычно советуют не выходить за рамки одной страницы, если ваш опыт работы не превышает десяти лет, или двух страниц, если вы более опытны. Почему так? Вот основные причины:

- Рекрутеры тратят на одно резюме в среднем не более 20 секунд. Если вы сократите размер и укажете в резюме только главные детали, их заметят. Большое резюме — бессмысленная вещь, никто его не будет внимательно читать.
- 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, вы просто не пробовали уложиться на страницу. Огромное 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 вашей трудовой деятельности. То, что вы продавали мороженое, не характеризует ваш интеллект или таланты в написании программ. В резюме нужно включать только значимые позиции.

Указывайте только значимые позиции

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

- «Благодаря моей реализации распределенного кэша было достигнуто сокращение времени прорисовки на 75 %, что в свою очередь привело к сокращению времени входа в систему» на 10 %.
- «Благодаря использованию windiff при реализации нового алгоритма сравнения средняя точность совпадений выросла с 1,2 до 1,5».

Конечно, не нужно пытаться формализовать все ваши достижения, но принцип, думаю, ясен. Нужно показать, что вы сделали, как вы сделали и какие результаты получены. В идеале вы должны сделать ваши достижения «измеряемыми».

Проекты

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

В список нужно включать по 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, но вы предстанете в лучшем свете.

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

Языки программирования и программные продукты

Программные продукты

Вообще-то говоря, я не рекомендую указывать в резюме умение работать с продуктами вроде Microsoft Office, — это должен знать каждый. Навыки работы с высокотехнологичными продуктами (Visual Studio, Linux) могут оказаться полезными, но, по большому счету, и они не имеют решающего значения.

Языки программирования

Знание языков программирования — хитрая штука. Что перечислять? Все языки, на которых когда-либо вам приходилось программировать, или только те, которые часто используете? Я советую перечислить большинство языков, которыми вы владеете, но обязательно укажите свой уровень опыта, например:

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

Если английский — не ваш родной язык

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

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

Часть V Подготовка к поведенческим вопросам

Поведенческие вопросы

Поведенческие вопросы задают, чтобы оценить вас как личность, уточнить резюме или даже освободить вас от собеседования. Так или иначе, эти вопросы важны и к ним следует подготовиться.

Как подготовиться

Поведенческие вопросы чаще всего формулируются в виде «Расскажите мне о времени, когда вы...» и могут затрагивать период жизни, когда вы работали над определенным проектом или занимали определенную должность. Я рекомендую подготовить и заполнить специальную таблицу.

Общие вопросы	Проект 1	Проект 2	Проект 3	Проект 4
Что вас стимулировало				
Чему вы научились				
Наиболее интересный момент				
Самая сложная ошибка				
От чего получили удовольствие				
Конфликты с членами команды				

В шапке таблицы перечислите основные пункты вашего резюме: проекты, задания или другую деятельность. Перечислите общие вопросы: что вам больше всего нравилось или не нравилось, что вы посчитали важным, чему научились, какая была самая серьезная ошибка и т. д. В каждой ячейке запишите соответствующую историю.

На собеседовании, когда вас будут спрашивать о каком-либо проекте, вы без труда вспомните соответствующий сюжет. Не забудьте взглянуть на эту таблицу перед собеседованием.

Я рекомендую сократить каждую историю до пары слов, которые можно легко вписать в ячейку, это упростит вашу подготовку.

Если вы проходите собеседование по телефону, таблица должна быть перед вашими глазами. Пары ключевых слов, записанных в каждой ячейке, хватит, чтобы активировать вашу память, вы легко и непринужденно сможете рассказать о любом проекте, что существенно лучше, чем пытаться прочитать вслух написанный ранее абзац.

Иногда полезно расширить таблицу «мягкими» темами — конфликты в команде, сбои в работе или трудные моменты, когда вам нужно было кого-либо переубедить. Такие темы выходят за пределы обязанностей рядового программиста, но если вы претендуете на позицию ведущего программиста, РМ или тестера, я советую подготовить дополнительную таблицу, охватывающую эти темы.

Вы отвечаете на поведенческие вопросы, не пытайтесь найти в памяти подходящий под вопрос ситуацию. Просто расскажите о себе, порассуждайте, как каждая история касалась лично вас.

Слабые места

Если вас спросят о слабых местах, расскажите о самом слабом месте. Ответы «Мое самое слабое место — я трудоголик», заставят интервьюера думать, что вы слишком много времени тратите на свою работу и не имеете собственного мнения о себе или не хотите признаваться в своих слабостях. Никто не захочет работать с таким человеком. Лучший ответ — сказать правду, указать ваше наименее слабое место, но продемонстрировать, что вы работаете и в скором времени исправите свои недостатки. Например: «Я бываю не очень внимателен к деталям. У меня есть и хорошая сторона — я быстро выполняю задания, но иногда все-таки делаю ошибки из-за невнимательности. Именно поэтому я по несколько раз проверяю полученный результат».

Что заставляет вас работать

Когда вам задают подобный вопрос, не нужно говорить: «Я хотел выучить много новых языков программирования и технологий». Этот ответ подойдет, только если вы действительно не знаете, что сказать. Интервьюер сразу решит, что проект был не для вас и сложный.

Что вопросы нужно задавать интервьюеру

Большинство интервьюеров дают вам шанс задать вопрос. Качество ваших вопросов, действительно или подсознательно, повлияет на их решение.

Некоторые вопросы могут возникнуть во время собеседования, но некоторые вы можете (и можете) подготовить заранее. Изучите историю и область деятельности компании или команды — это поможет вам подготовить свои вопросы.

Вопросы можно разделить на несколько категорий.

Интервьюющие вопросы

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

Сколько времени ежедневно вы тратите на программирование?

Сколько встреч вы проводите каждую неделю?

Какое количественное соотношение между тестерами, разработчиками и менеджерами программ? Как они взаимодействуют? Как происходит планирование проекта?

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

Принципиальные вопросы

Эти вопросы предназначены для демонстрации ваших знаний программирования и технологий, а также говорят о вашем отношении к компании или продукту:

Я заметил, что вы используете технологию X. Как вы решаете проблему Y?

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 исследовать 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. Посоветуйте, где можно узнать об этом?
2. Я не знаком с технологией X, но слышал, что это очень интересное решение. Не могли бы вы мне рассказать, как она работает?

Ответы на поведенческие вопросы

Собеседования обычно начинаются и заканчиваются непринужденной беседой. Это время, когда интервьюер может задать вопросы о вашем резюме, а вы можете задать вопросы интервьюеру. Данная часть собеседования позволяет лучше узнать вас, а вам дает возможность расслабиться.

Запомните несколько советов, они пригодятся, когда вы отвечаете на вопросы.

Отвечайте четко, но без высокомерия

Высокомерие — это красная карточка, но вы же хотите выглядеть достаточно впечатльно. Как этого добиться и не показаться высокомерным? Четко формулируйте свои мысли! Рассмотрим пример:

- *Кандидат 1:* «Именно я делал всю самую сложную работу для команды».
- *Кандидат 2:* «Я занимался реализацией файловой системы — наиболее важный компонент, поскольку...»

Кандидат 2 выглядит не только более впечатльным, но и менее высокомерным.

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. Ограничите подробности, оставьте только ключевые пункты, сделайте рассказ легким для восприятия. Вот небольшой пример: «При исследовании наиболее 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. Я могу рассказать подробнее, если вам это интересно». Это демонстрирует ключевые моменты, но не утомляет интервьюера.

Структурируйте ответ

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

«Золотой самородок»

Техника «золотой самородок» предполагает, что вы сразу выкладываете перед интервьюером «самородок», который кратко описывает, о чём будет ваш ответ. 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 захватить внимание интервьюера, а ему — мгновенно понять, о чем будет ваш рассказ. Вы сразу демонстрируете свои коммуникативные навыки.

SAR

Техника 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 взаимодействии 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 изо всех сил пытался избежать дискуссий.

Действие: Однажды после лекции я отвел его в сторону, чтобы поговорить о курсе, а потом плавно перешел к разговору о проекте. Сразу выяснилось, что он — стеснительный человек и ему не хватает уверенности в своих силах. При дальнейшей совместной работе я учел этот факт и стал его хвалить, чтобы поднять его самооценку.

Результат: Хотя этот человек оставался слабым звеном в команде, но он стал работать намного лучше. Он делал всю возложенную на него работу и принимал участие в обсуждениях проекта. Я считаю, что у нас получилась настоящая командная работа над проектом.

Описания ситуации и результата должны быть очень краткими. Ваш интервьюер не нуждается в избыточных подробностях. При использовании модели SAR интервьюер легко понимает, какой была ситуация, каковы наши действия и что получилось в результате.

Часть VI Технические вопросы

Подготовка

Если вы купили эту книгу, то, скорее всего, уже проделали достаточно долгий путь к вершине отличной технической подготовки. Поздравляю, прекрасная работа!

Но существуют как лучшие, так и худшие способы подготовки. Многие кандидаты считают, что достаточно просмотреть задачи и готовые решения. Это в чем-то похоже на попытку научиться считать, читая книги. Но если вы хотите научиться решать задачи, запоминание готовых решений не поможет.

Как организовать подготовку

Для каждой задачи из этой книги:

- *Попытайтесь решить задачу самостоятельно.* Я имею в виду действительно попытаться решить задачу. Вам встретится множество сложных заданий — и это нормально. Когда вы решите задачу, подумайте об эффективности использования пространства (памяти) и времени выполнения. Задайтесь вопросом, можно ли ускорить выполнение программы, оптимизируя использование памяти, и наоборот.
- *Запишите код алгоритма на бумаге.* Вы всю жизнь программируете на компьютере, и это, безусловно, хорошо. Но на собеседовании вам не поможет ни подсветка синтаксиса, ни автозавершение кода, ни компиляция. Сымитируйте условия собеседования, записывая код на бумаге.
- *Протестируйте свой код.* Представьте, что вы — компилятор, проверьте код на наличие ошибок. Вам придется это делать на собеседовании, поэтому лучше подготовиться заранее.
- *Ведите написанный код в компьютер «как есть».* Возможно, вы обнаружите множество ошибок. Проанализируйте все ошибки и сделайте все, чтобы на настоящем собеседовании их не допустить.

Очень полезны псевдоинтервью. На CareerCup.com вы найдете примеры псевдоинтервью с сотрудниками Microsoft, Google, Amazon — используйте их, когда будете практиковаться с друзьями. Хотя ваши друзья не являются профессиональными интервьюерами, они в состоянии проверить решения задач на программирование и алгоритмизацию.

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.

Структуры данных	Алгоритмы	Концепции
Связные списки	Поиск в ширину	Манипуляции битами
Бинарные деревья	Поиск в глубину	Одиночка (шаблон проектирования)
Графы	Бинарный поиск	Фабричный шаблон проектирования

Структуры данных	Алгоритмы	Концепции
Стеки	Сортировка слиянием	Память (стек vs куча)
Очереди	Быстрая сортировка	Рекурсия
Векторы/Списки массивов	Вставка в дерево, поиск и т. д.	Время порядка «O-большое» ¹
Хэш-таблицы		

Убедитесь, что вы понимаете, как использовать и реализовывать каждую из задач, знаете их область применения, эффективность использования памяти и время выполнения. Вам могут задать вопрос по теме из таблицы или попросить реализовать такую-либо ее модификацию.

Обратите внимание на хэш-таблицы — это наиболее важная тема. Она часто встречается на собеседованиях.

Таблица степеней двойки

Некоторые люди помнят ее как таблицу умножения, если вы не относитесь к их числу, вам нужно подготовиться. Таблица степеней двойки пригодится в задачах масштабируемости — в расчетах необходимого объема памяти для набора данных.

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

Используя эту таблицу, вы можете легко рассчитать, например, хватит ли имеющегося объема памяти для хэш-таблицы, отображающей каждое 32-битное число в булевое значение.

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

Нужно ли знать все о программировании на C++, Java или других языках

Хотя я лично никогда не любила вопросы такого рода (например: «Что такое vtable?»), многие интервьюеры действительно их задают. В крупных компаниях — Microsoft, Google, Amazon и других — таким вопросам не уделяется слишком много внимания. Вы должны понимать основные концепции языка, которым вы владеете, но лучше сфокусироваться на структурах данных и алгоритмах.

¹ Про «O-большое» можно прочитать в Википедии http://ru.wikipedia.org/wiki/«O»_большое_и_«o»_малое. — Примеч. ред.

В то же время для небольших компаний вопросы, связанные с языками программирования, могут быть очень важны. Поиските компанию, в которой собираетесь проходить собеседование, на CareerCup.com. Если вашей компании там нет, поиските аналогичную компанию. Большинство стартапов проверяют навыки именно «их» языка программирования.

Ответы на технические вопросы

Собеседование не может быть простым. Если вам не удается дать ответ «с ходу» — это нормально! Только десять из более чем ста двадцати человек смогли быстро разобраться в моих любимых задачах.

Если вам достался сложный вопрос, не нужно паниковать. Начните с планирования решения — покажите интервьюеру, что вы не застряли.

Запомните еще одно правило — вы не должны останавливаться, пока интервьюер не скажет, что решение закончено. Что я имею в виду? Если вы уже придумали алгоритм, продолжите рассуждать о возможных ошибках в его работе. Если вы пишете программный код, попытайтесь найти в нем ошибки. Если вы относитесь к числу тех самых 110 кандидатов, у вас, вероятно, будут ошибки.

Пять шагов к решению

Любую техническую задачу на собеседовании можно решить за пять шагов:

1. Задайте интервьюеру вопросы, чтобы снять неоднозначности.
2. Разработайте алгоритм.
3. Запишите псевдокод, но сообщите интервьюеру, что вы намерены написать решение на конкретном языке программирования.
4. В умеренном темпе начните писать программный код.
5. Проверьте написанную программу и внимательно исправьте ошибки.

Остановимся на каждом шаге поподробнее.

Шаг 1. Задайте вопросы

Технические задачи более неоднозначны, чем может показаться на первый взгляд, убедитесь, что вам ясна суть вопроса. Задача может оказаться намного проще, чем казалась в начале. Некоторые интервьюеры (в частности, в Microsoft) специально проверяют, поняли ли вы задачу.

Вот несколько примеров хороших вопросов: какие типы данных используются? Какие объемы данных будут использоваться? Какие исходные предположения нужны для решения? Кто будет пользователем?

Задача: «Разработайте алгоритм сортировки списка»

Вопрос: Какой список нужно сортировать? Массив? Связный список?

Ответ: Массив.

Вопрос: Что будет в массиве? Числа? Символы? Строки?

Ответ: Числа.

Вопрос: Числа будут целыми?

Ответ: Да.

Вопрос: Что представляют собой эти числа? Это идентификаторы? Значение чего-либо?

Ответ: Это возраст клиентов.

Вопрос: Сколько будет клиентов?

Ответ: Миллион.

Теперь постановка задачи изменилась: нужно отсортировать массив из миллиона целых чисел в диапазоне от 0 до 130 (максимально возможный возраст). Как ее решить? Достаточно создать массив из 130 элементов и посчитать количество значений для каждого возраста.

Шаг 2. Разработайте алгоритм

Разработайте алгоритм, но при этом помните о пяти подходах алгоритмизации (см. следующий раздел). Пока вы обдумываете алгоритм, не забудьте ответить на вопросы:

- Сколько памяти и времени понадобится для реализации этого алгоритма?
- Что произойдет, если данных будет больше, чем запланировано?
- Какие проблемы могут возникнуть в процессе работы вашего алгоритма? Например, если вы создаете модификацию бинарного дерева поиска, как ваш алгоритм повлияет на время вставки, поиска и удаления?
- Какие компромиссные решения возможны с учетом существующих ограничений? Для каких сценариев компромиссное решение будет наименее оптимальным?
- Нужна ли дополнительная исходная информация (в нашем случае — возраст клиентов или порядок сортировки) для реализации алгоритма? Обычно интервьюер специально предоставляет вам дополнительную информацию.

Метод грубой силы — вполне приемлемый и даже рекомендованный путь. После его разработки вы можете провести оптимизацию. Рано или поздно вы придете к оптимальному решению, но это не означает, что оно будет первым пришедшим в голову.

Шаг 3. Псевдокод

Псевдокод поможет в общих чертах набросать ваши соображения и избежать большей части ошибок. Но не забудьте сообщить интервьюеру, что сейчас вы пишете псевдокод, а затем перейдете к языку программирования. Много кандидатов используют псевдокод, чтобы избежать написания программы, но вы же не хотите быть одним из них?

Шаг 4. Код

Не нужно торопиться, чтобы потом не было мучительно больно. Пишите программный код спокойно и аккуратно. Запомните советы:

- Используйте структуры данных везде, где это возможно; выберите подходящую структуру данных или разработайте собственную. Например, если вас просят определить минимальный возраст группы людей, можно создать специальную структуру данных — Person. Этим вы покажете интервьюеру, что способны создать хороший объектно-ориентированный проект.
- Не создавайте очень длинные программы. Когда вы записываете свой код на доске, начинайте с верхнего левого угла, а не с середины, чтобы ответ поместился целиком.

Шаг 5. Проверка

Вам нужно протестировать код. В первую очередь обратите внимание на следующие моменты:

- экстремальные случаи — 0, отрицательное значение, `null`, максимумы, минимумы;
- ошибки пользователя — что случится, если пользователь передаст `null` или отрицательное значение;
- общие случаи — протестируйте типовое поведение программы.

Если алгоритм сложный или исключительно численный (смещение, булева алгебра и т. д.), тестируйте код по мере написания, а не по завершении работы.

Если вы находите ошибки (а они будут), не торопитесь их исправлять — попытайтесь найти причину их возникновения. Вы же не хотите стать одним из кандидатов, который, обнаруживая, что функция возвращает `true` вместо `false`, просто инвертирует ее значение. Это устранит ошибку в конкретном случае, но неизбежно породит новые. Работая над ошибками, задумайтесь, почему код перестал работать. Это поможет написать красивый и чистый код намного быстрее.

Пять подходов к алгоритмизации

Не существует универсального суперспособа, решающего любую задачу алгоритмизации, но приведенные далее подходы могут оказаться полезны. Помните: чем больше задач вы решите, тем проще будет выбрать подходящий способ.

Пять способов, приведенных далее, можно использовать по отдельности или вместе. Попробуйте подход «Упростить и обобщить», а затем перейдите к «Сопоставлению с образцом».

Подход 1. Приводим пример

Мы начнем со способа, с которым вы, вероятно, знакомы, даже если никогда не обращали на него внимания. Напишите несколько примеров задачи, и вы увидите, можно ли получить общее правило из них.

Пример: задано время, нужно рассчитать угол между часовой и минутной стрелками.

Давайте начнем, пусть исходное время — 3:27. Мы можем нарисовать циферблат, установить часовую стрелку на 3 часа, а минутную — на 27 минут.

Введем следующие обозначения: h — это часы, m — минуты. Предположим, что h может принимать значения в диапазоне от 0 до 23 включительно.

Теперь можно сформулировать следующие правила:

- угол между минутной стрелкой и «полуднем»: $360 * m / 60$;
- угол между часовой стрелкой и «полуднем»: $360 * (h \% 12) / 12 + 360 * (m / 60) * (1 / 12)$;
- угол между часовой стрелкой и минутной: (угол часовой стрелки — угол минутной стрелки) % 360.

Окончательное выражение можно упростить до $30h - 5.5m$.



Подход 2. Сопоставление с образцом

«Сопоставление с образцом» подразумевает, что мы сравниваем задачу с подобной и пытаемся приспособить алгоритм для нашего случая.

Пример. Отсортированный массив был циклически сдвинут так, что элементы оказались в следующем порядке: 3 4 5 6 7 1 2. Как найти минимальный элемент? Вы можете исходить только из предположения, что элементы в массиве не повторяются.

Существуют две задачи, которые можно рассматривать как аналог:

- Поиск минимального элемента в массиве.
- Поиск определенного элемента в сортированном массиве (бинарный поиск).

Поиск минимального элемента в массиве не самый интересный алгоритм — нужно пройтись по всем элементам. Вряд ли он будет полезен.

А вот бинарный поиск можно использовать. Вы знаете, что массив был отсортирован, а затем циклически сдвинут. Поэтому значения увеличиваются, а затем сбрасываются в исходную точку и снова растут. Минимальный элемент оказывается в точке сброса.

Если вы сравните средний и последний элементы (6 и 2), то узнаете, что точка сброса должна находиться между этими значениями ($\text{mid} > \text{right}$). Но это может произойти, только если массив сбрасывался между данными значениями.

Если mid меньше, чем right , значит, точка сброса находится в левой части массива, либо точки сброса не существует (массив отсортирован). Так или иначе, минимальный элемент находится там.

Мы можем использовать этот подход, деля массив пополам, подобно алгоритму бинарного поиска, и, в конечном счете, найдем минимальный элемент (или точку сброса).

Подход 3. Упростить и обобщить

Этот алгоритм подразумевает многошаговый подход. Во-первых, мы изменяем ограничение (тип данных или количество исходных данных), чтобы упростить задачу. Затем мы решаем упрощенную версию задачи. Как только алгоритм для упрощенной задачи получен, мы обобщаем ее и пытаемся приспособить полученное решение к более сложной версии.

Пример. Требование о выкупе было склеено из вырезанных из газеты отдельных слов. Как проверить, что требование о выкупе (представленное в виде строки) было сделано с использованием конкретной газеты (строки)?

Для упрощения задачи предположим, что из газеты вырезались отдельные буквы, а не слова.

Чтобы решить упрощенную задачу, создадим массив и подсчитаем символы. Каждый элемент массива соответствует одной букве. Сначала мы подсчитываем, сколько раз повторяется каждый знак в требовании о выкупе, а затем проверяем, есть ли в газете все эти символы.

При обобщении алгоритма используется тот же подход. Только вместо создания массива с количеством символов можно создать хэш-таблицу, которая сопоставляет слова с частотой их использования.

Подход 4. Базовый случай и сборка решения

Данный метод идеально подходит для решения целого ряда задач. Мы можем решить задачу для базового случая ($n = 1$). Это обычно означает запись корректного результата. Затем решить задачу для $n = 2$, учитывая, что ответ для $n = 1$ уже найден. Затем заняться случаем $n = 3$, учитывая, что ответы для $n = 1$ и $n = 2$ известны.

В итоге мы можем построить решение, которое позволит найти результат для N , если известен правильный результат для $N - 1$. Каждый раз наше решение основывается на предыдущем результате.

Пример. Разработайте алгоритм для вывода всех возможных перестановок символов в строке (считайте, что все символы используются только один раз).

Дана тестовая строка — abcdefg.

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

Вот и первый «интересный» случай. Можем ли мы сгенерировать $P("abc")$, если у нас есть ответ $P("ab")$? Итак, у нас появляется дополнительная буква («с») 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"}
```

Мы разобрались с шаблоном и можем разработать общий рекурсивный алгоритм. Сгенерируйте все перестановки строки $s_1 \dots s_n$, удалив последний символ (для $s_1 \dots s_{n-1}$). Получив список всех перестановок $s_1 \dots s_{n-1}$, последовательно пройдитесь по каждому элементу-строке списка, добавляя символ s_n в каждую позицию строки.

Данный подход позволяет создавать рекурсивные алгоритмы.

Подход 5. Мозговой штурм структур данных

Данный способ трудно назвать идеальным, но зачастую он срабатывает. Мы просто проходим по списку структур данных и пытаемся 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. Давайте отложим эту структуру данных и вернемся к ней при необходимости.
- Бинарное дерево? Вполне возможно, бинарные деревья подходят для задач сортировки. Мы можем 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 min-куче, так что самый маленький элемент оказывается в вершине, а «меньшая» половина — в max-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 понять, какой подход более применим к той или иной задаче.

Как выглядит хороший код

Вы, возможно, неоднократно слышали, что работодатели хотят видеть «хороший» и «чистый» код. Но что это такое и как продемонстрировать его на собеседовании?

О хорошем коде всегда можно сказать, что он:

- **правильный:** код корректно обрабатывает все корректные и некорректные входные данные;
- **эффективный:** код максимально эффективен с точки зрения времени и пространства (памяти). Эффективность включает асимптотическую («O-большое») и практическую (реальную) эффективность. При расчете зависимости времени выполнения программы от ее сложности постоянный коэффициент можно отбросить, но в реальной жизни он может оказать влияние на эффективность;
- **простой:** если вы можете реализовать алгоритм в десяти строках, не пишите сто. Создайте код максимально быстро;
- **читаемый:** другой разработчик должен прочитать ваш код и понять, что и как делается. Для читаемого кода необходимы комментарии, они делают код более понятным. Сдвига строк недостаточно;
- **обслуживаемый:** код должен легко адаптироваться к изменениям, которые возникают на протяжении жизненного цикла продукта, он должен обслуживаться другими программистами так же легко, как и разработчиком.

Следование всем этим правилам требует определенных компромиссов. Например, стоит принести в жертву немного эффективности, но сделать код более удобным в сопровождении и наоборот.

Вы должны думать обо всех этих аспектах, когда пишете программный код на собеседовании.

Структуры данных

Предположим, что вас попросили написать функцию, выполняющую сложение двух простых полиномов, представленных в виде $Ax^a + Bx^b + \dots$. Интервьюер не хочет, чтобы вы делали парсинг строк, ему нужно, чтобы вы использовали структуру данных, способную хранить полином.

Есть много разных способов решить такую задачу.

Неудачная реализация

Плохая реализация подразумевает хранение полинома в виде массива чисел с двойной точностью, где k -й элемент соответствует элементу x^k . Такая структура может стать причиной ошибок при необходимости представить полином, содержащий отрицательные или дробные степени. Кроме того, для хранения полинома, содержащего только один член x^{1000} , потребуется массив из 1000 элементов.

```
1 int[] sum(double[] poly1, double[] poly2) {
2     ...
3 }
```

Чуть лучшая реализация

Чуть лучшая реализация подразумевает хранение полинома в двух массивах — `coefficients` и `exponents`. Полином в этом случае может храниться в любом порядке, но i -й член полинома должен иметь вид $\text{coefficients}[i] * x^{\text{exponents}[i]}$.

Таким образом, если $\text{coefficients}[p] = k$ и $\text{exponents}[p] = m$, то p -й член будет равен kx^m . Хотя данная реализация не имеет таких ограничений, как предыдущее решение, она все еще далека от идеала. Мы должны использовать пару массивов для каждого полинома. Если массивы будут разной длины, то в полиноме оказываются «неопределенные» значения. Да и результат будет не очень удобным, так как при вызове функция будет возвращать два массива.

```
1 ??? sum(double[] coeffs1, double[] expon1,
2         double[] coeffs2, double[] expon2) {
3     ...
4 }
```

Хорошая реализация

Хорошая реализация — разработка собственной структуры данных для хранения полинома.

```
1 class PolyTerm {
2     double coefficient;
3     double exponent;
4 }
5
6 PolyTerm[] sum(PolyTerm[] poly1, PolyTerm[] poly) {
7     ...
8 }
```

Кое-кто будет утверждать, что это «сверхоптимизация». Возможно, да, а возможно — нет. Независимо от вашего мнения, это решение продемонстрирует, что вы думаете о коде и не пытаетесь решить задачу самым простым (и самым быстрым) способом.

Обоснованное многократное использование кода

Предположим, что вас попросили написать функцию, проверяющую на равенство двоичное и шестнадцатеричное представление числа, хранимое в виде строки. Изящная реализация этой задачи подразумевает повторное использование кода:

```
1 public boolean compareBinToHex(String binary, String hex) {
2     int n1 = convertToBase(binary, 2);
```

```

3     int n2 = convertToBase(hex, 16);
4     if (n1 < 0 || n2 < 0) {
5         return false;
6     } else {
7         return n1 == n2;
8     }
9 }
10
11 public int digitToValue(char c) {
12     if (c >= '0' && c <= '9') return c - '0';
13     else if (c >= 'A' && c <= 'F') return 10 + c - 'A';
14     else if (c >= 'a' && c <= 'f') return 10 + c - 'a';
15     return -1;
16 }
17
18 public int convertToBase(String number, int base) {
19     if (base < 2 || (base > 10 && base != 16)) return -1;
20     int value = 0;
21     for (int i = number.length() - 1; i >= 0; i--) {
22         int digit = digitToValue(number.charAt(i));
23         if (digit < 0 || digit >= base) {
24             return -1;
25         }
26         int exp = number.length() - 1 - i;
27         value += digit * Math.pow(base, exp);
28     }
29     return value;
30 }
```

Возможно, вы смогли бы написать отдельный код, преобразующий двоичное число в шестнадцатеричное, но это сделает нашу программу более громоздкой и тяжелой в обслуживании. Поэтому мы используем код повторно, вызывая общие методы `convertToBase` и `digitToValue`.

Модульность

Написание модульного кода подразумевает выделение изолированных блоков программы в отдельные методы. Это делает код более удобным, читаемым и тестируемым.

Допустим, что вам нужно написать код, меняющий местами минимальный и максимальный элементы в целочисленном массиве. Этую задачу можно реализовать в одном методе.

```

1 public void swapMinMax(int[] array) {
2     int minIndex = 0;
3     for (int i = 1; i < array.length; i++) {
4         if (array[i] < array[minIndex]) {
5             minIndex = i;
6         }
7     }
8 }
```

продолжение ➔

```

9     int maxIndex = 0;
10    for (int i = 1; i < array.length; i++) {
11        if (array[i] > array[maxIndex]) {
12            maxIndex = i;
13        }
14    }
15
16    int temp = array[minIndex];
17    array[minIndex] = array[maxIndex];
18    array[maxIndex] = temp;
19 }

```

Но эту же задачу можно решить с использованием модульного кода, выделив относительно изолированные блоки кода в отдельные методы.

```

1 public static int getMinIndex(int[] array) {
2     int minIndex = 0;
3     for (int i = 1; i < array.length; i++) {
4         if (array[i] < array[minIndex]) {
5             minIndex = i;
6         }
7     }
8     return minIndex;
9 }
10
11 public static int getMaxIndex(int[] array) {
12     int maxIndex = 0;
13     for (int i = 1; i < array.length; i++) {
14         if (array[i] > array[maxIndex]) {
15             maxIndex = i;
16         }
17     }
18     return maxIndex;
19 }
20
21 public static void swap(int[] array, int m, int n) {
22     int temp = array[m];
23     array[m] = array[n];
24     array[n] = temp;
25 }
26
27 public static void swapMinMaxBetter(int[] array) {
28     int minIndex = getMinIndex(array);
29     int maxIndex = getMaxIndex(array);
30     swap(array, minIndex, maxIndex);
31 }

```

Немодульный код не так уж и плох, но модульный код значительно легче протестировать, так как каждый его компонент можно проверить по отдельности. Чем код сложнее, тем оправданнее его модульность. Такой подход облегчит чтение и поддержку кода. Ваш интервьюер хочет увидеть, насколько вы владеете этими навыками.

Гибкость и надежность

Когда интервьюер просит написать код для проверки игрового поля для игры в крестики-нолики, это еще не означает, что доска имеет размер 3×3. Почему бы сразу не написать универсальный код для доски произвольного размера $N \times N$?

Гибкий и универсальный код предполагает использование констант вместо переменных или шаблонов/обобщений, позволяющих решить задачу. Если мы можем написать код, решающий задачу в общем виде, мы должны это сделать.

Конечно, всему есть предел. Если решение для общего случая оказывается слишком сложным и громоздким, то лучше ограничиться простым случаем, который фигурирует в задании.

Проверка

Отличительной чертой предусмотрительного программиста является то, что он не делает предположений о входных данных. Вместо этого он проверяет введенную последовательность с помощью операторов `ASSERT` или `if`. Вернемся к написанному ранее коду, преобразующему числа из системы счисления по основанию 1 (двоичной или шестнадцатеричной системы) в целое число (`int`).

```
1 public int convertToBase(String number, int base) {  
2     if (base < 2 || (base > 10 && base != 16)) return -1;  
3     int value = 0;  
4     for (int i = number.length() - 1; i >= 0; i--) {  
5         int digit = digitToValue(number.charAt(i));  
6         if (digit < 0 || digit >= base) {  
7             return -1;  
8         }  
9         int exp = number.length() - 1 - i;  
10        value += digit * Math.pow(base, exp);  
11    }  
12    return value;  
13 }
```

В строке 2 мы проверяем, корректно ли выбрана система счисления (если основание больше 10, то это шестнадцатеричная система). В строке 6 мы производим еще одну проверку: нужно убедиться, что каждая цифра находится в пределах допустимого диапазона.

Подобные проверки очень важны в реальных программах, а значит, их оценят и на собеседовании.

Конечно, проверка ошибок — утомительное занятие и на собеседовании отнимает много драгоценного времени. Но вам важно продемонстрировать свое умение сделать обработку ошибок. Если требуемая проверка на ошибки оказывается сложнее, чем оператор `if`, лучше оставить свободное место на листке и сказать интервьюеру, что собираетесь заполнить его кодом проверки ошибок, когда закончите основную часть программы.

Часть VII Жизнь после собеседования

Реакция на предложение и на отказ

Стоит только вам подумать, что после собеседования можно расслабиться, как на вас обрушаются новые переживания и волнения. Должны ли вы согласиться с предложением? Правилен ли ваш выбор? Как отказаться от работы? Есть ли время для принятия решения? Сейчас мы поговорим о некоторых тонкостях, которые позволят вам оценить и обсудить предложение.

Сроки принятия решения

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

Вы отказываетесь от работы

Форма, в которой вы отказываетесь от предложения, имеет очень большое значение. Даже если работа в этой компании на данный момент вам не интересна, 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. 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 они без обид согласятся с вашей аргументацией.

Вам отказали

Крупные ИТ-компании отклоняют 80 % кандидатов по причине недостаточного профессионализма оных. Зачастую позднее те же компании инициируют повторное собеседование с ранее отклоненным претендентом.

Воспринимайте неприятный звонок как временную неудачу, а не как приговор. Поблагодарите своего рекрутера за потраченное время, объясните, что вы разочарованы, но понимаете их позицию, и поинтересуйтесь, когда можно будет пройти собеседование повторно.

Причины отказа не всегда понятны. Рекрутеры вряд ли озвучат истинную причину, но наверняка ваши шансы могли быть выше, если бы вы лучше подготовились, поэтому вместо поиска причины отказа сфокусируйтесь на подготовке. Вы можете проанализировать причину, но мой опыт говорит, что кандидаты очень редко могут должным образом оценить свои результаты. Вы предполагаете, что превосходно ответили на вопросы, но все относительно. Помните, что большинство отказов происходит из-за недостаточных навыков программирования и алгоритмизации, поэтому сфокусируйтесь на этом.

Вам сделали предложение

Поздравляю, вы получили предложение о работе! А если вы счастливчик, то, возможно, даже получили несколько предложений. Задача вашего рекрутера — убе-

принять его предложение. Но достойна ли вас эта компания? Давайте

Финансовый пакет

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 самым 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 20–30 % дороже, чем в Сиэтле (в том числе из-за десятипроцентного Калифорнийского налога). Используйте онлайн-ресурсы, чтобы оценить этот фактор;

ежегодная премия. Ежегодная премия в IT-компании может варьироваться от 3 до 30 %. Ваш рекруттер может сообщить вам такую информацию, но если он отвечает, поищите знакомых, работающих в этой компании;

фондовые опционы и гранты. Акции являются большей частью ежегодной компенсации. Подобно льготам и премиям, расчет фондовой компенсации нужно производить в расчете на три года, затем полученное значение добавлять к вашей зарплате.

Не менее помните, что также нужно учитывать перспективы карьерного роста, а это может иметь большее значение для ваших долгосрочных финансовых планов, чем зарплата. Думайте прежде, чем делать ставку на озвученную заработную плату.

Карьера рост

Вы несколько взволнованы, поскольку вам только что сделали заманчивое предложение, но наверняка 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 вы будете работать на Microsoft в Кремниевой долине и захотите уйти, то легко сможете устроиться

в любую другую компанию. Но если вы будете работать на Microsoft в Сиэтле, то будете ограничены только Google, Amazon и немногочисленными небольшими компаниями. Если вы работаете в AOL в Далласе, ваши возможности по переходу в другую компанию еще более ограничены. У вас просто не будет выбора — если в городе нет других IT-компаний, вы будете вынуждены или оставаться, или готовиться к переезду.

Стабильность компании

Тут все зависит от случая, но я не рекомендую зацикливаться на этом аспекте. Если вам придется уйти, то вы без проблем найдете работу в аналогичной компании. Ответьте сами себе: что произойдет, если вас уволят? Каковы ваши перспективы найти новую работу?

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? Вы сможете с ними работать?
- *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 цейтнота (*deadline*) обычно гораздо выше стандартной.

Дополнительно выясните, возможен ли переход из одной команды в другую (как, например, в Google). 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. Но рекрутеры не будут отзывать предложение только потому, что вы пытаетесь добиться для себя некоторых преимуществ, поэтому вам нечего терять.

2. *Всегда имейте запасной вариант.* Рекрутеры 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 \$7000, чем просто просите ее увеличить, — вам могут предложить \$1000, и это будет формальным шагом к удовлетворению ваших требований.
4. *Завышайте требования.* При переговорах люди не всегда получают желаемое. Просите больше, чем надеетесь получить, может, вам повезет и вы получите «компромиссную» прибавку к зарплате.
5. *Думайте не только о зарплате.* Компании часто готовы пойти на любые 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, поскольку можно сэкономить и на переезд потратить меньшую сумму.
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. Во время переговоров важно чувствовать себя комфортно.

Кроме того, если вы ведете переговоры с крупной компанией, то в ней могут существовать уровни градации служащих. Всем служащим одного уровня платят одинаково. В Microsoft, например, существует такая система уровней. Вы можете торговаться в пределах зарплаты служащих вашего уровня, но для большего требуется повышение уровня. Вам придется убедить начальника 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, все 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, переживаете. 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 вы наслаждаетесь работой, очень легко забыться и перестать понимать, что ваша карьера забуксовала. Чтобы такого не случилось, представляйте свою дальнейшую карьеру (хотя бы в общих чертах) прежде, чем начнете 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. Онлайн-дружба — это хорошо, но очное знакомство всегда лучше. Установите дружеские отношения со своим непосредственным руководителем и коллегами. Даже если вы уйдете из компании, контакты со старой командой останутся.

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 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.

Помните, что вы можете устроиться на работу в любой момент. Трудно, конечно, неизвестно заранее. Но зная, что могут вас ждать дальше, вы можете спланировать будущее. Идея о том, что вы можете устроиться на работу в любой момент, может быть полезна, если вы не уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время. Но если вы уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время, то это может быть полезно для вас.

Помните, что вы можете устроиться на работу в любой момент. Трудно, конечно, неизвестно заранее. Но зная, что могут вас ждать дальше, вы можете спланировать будущее. Идея о том, что вы можете устроиться на работу в любой момент, может быть полезна, если вы не уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время. Но если вы уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время, то это может быть полезно для вас.

Переговоры

Вы можете устроиться на работу в любой момент. Трудно, конечно, неизвестно заранее. Но зная, что могут вас ждать дальше, вы можете спланировать будущее. Идея о том, что вы можете устроиться на работу в любой момент, может быть полезна, если вы не уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время. Но если вы уверены в том, что у вас есть реальная возможность устроиться на работу в ближайшее время, то это может быть полезно для вас.

Часть VIII Вопросы собеседования

Присоединяйтесь к нам на www.CrackingTheCodingInterview.com и загружайте полные, совместимые с Java/Eclipse решения, читайте обсуждения задач из этой книги с другими пользователями, отправляйте нам свои отзывы, просматривайте список ошибок и опечаток, отправляйте свое резюме и обращайтесь за дополнительными советами.

СТРУКТУРЫ ДАННЫХ. ВОПРОСЫ И СОВЕТЫ

1. Массивы и строки

Надеюсь, что все читатели знакомы с массивами и строками, поэтому нет необходимости вникать в детали, лучше мы сфокусируемся на некоторых общих методах работы с этими структурами данных.

Обратите внимание, что вопросы, относящиеся к массивам и к строкам, одинаковы. Таким образом, любой вопрос о массиве можно рассматривать как вопрос о строке и наоборот.

Хэш-таблицы

Хэш-таблица — это структура данных, связывающая ключи со своими значениями для более эффективного поиска. В самом простом случае хэш-таблица представляет собой совокупность обычного массива и хэш-функции. Когда вам нужно вставить объект и его ключ, хэш-функция преобразует ключ в число, которое является индексом массива. Объект хранится по полученному индексу.

Такая реализация, скорее всего, не будет работать правильно — значение ключа должно быть уникальным, иначе мы можем случайно перезаписать данные. В результате полученный массив окажется громадным: его размер должен превышать суммарный размер всех ключей, только так можно избежать «коллизий».

Вместо огромного массива, заполняемого индексируемыми объектами `hash(key)`, мы можем создать массив меньшего размера и хранить объекты в связанном списке по индексу `hash(key) % array_length`. Чтобы получить объект с определенным ключом, нам придется произвести поиск по связанному списку.

Существует и другой способ реализовать хэш-таблицу — *бинарное дерево поиска*. В этом случае можно обеспечить время поиска $O(\log N)$, если дерево будет сбалансированным. Кроме того данный подход позволит использовать гораздо меньше пространства (памяти), поскольку нет необходимости сразу создавать огромный массив.

Попрактикуйтесь в реализации хэш-таблиц перед собеседованием. Это одна из наиболее любимых интервьюерами структур данных.

Вот простейший пример работы с хэш-таблицей на языке Java:

```
1 public HashMap<Integer, Student> buildMap(Student[] students) {
2     HashMap<Integer, Student> map = new HashMap<Integer, Student>();
3     for (Student s : students) map.put(s.getId(), s);
4     return map;
5 }
```

Обратите внимание, что хэш-таблицы не всегда являются идеальным решением. Использовать их или нет, решать вам — все зависит от поставленной задачи.

ArrayList (динамический массив)

`ArrayList` — это массив, размер которого может изменяться во время выполнения программы; он обеспечивает время доступа $O(1)$. Типичная реализация — при заполнении массива его размеры увеличиваются в два раза. Каждое удвоение занимает $O(n)$ времени, но поскольку данное событие происходит довольно редко, считается, что время доступа к массиву остается $O(1)$.

```
1 public 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 }
```

StringBuffer (буфер строк)

Представьте, что вам нужно объединить несколько строк. Каким будет время выполнения программы? Рассмотрим n строк одинаковой длины (x).

```
1 public 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 $O(x + 2x + \dots + nx) = O(xn^2)$.

`StringBuffer` поможет вам избавиться от этой проблемы. Мы просто создаем массив всех строк и копируем их в строку только в случае необходимости.

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

Изученным упражнением станет создание собственной версии `StringBuffer` с использованием строк, массивов и общих структур данных.

Вопросы интервью

1 Реализуйте алгоритм, определяющий, все ли символы в строке встречаются один раз. При выполнении этого задания нельзя использовать дополнительные структуры данных.

2 Реализуйте функцию `void reverse(char* str)` на С или C++. Функция должна циклически сдвигать строку, заканчивающуюся символом `null`.

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

4 Напишите метод, заменяющий все пробелы в строке символами '`%20`'. Можно предположить, что длина строки позволяет сохранить дополнительные символы и «истинная» длина строки известна. (Примечание: при реализации метода на Java используйте символьный массив.)

Пример:

Ввод: "Mr John Smith "

Выход: "Mr%20John%20Smith"

5 Реализуйте метод, осуществляющий сжатие строки, на основе счетчика повторяющихся символов. Например, строка `aabccccc aa` должна превратиться в `a2b1c5a3`. Если «сжатая» строка оказывается длиннее исходной, метод должен вернуть исходную строку.

6 Дано: изображение в виде матрицы размером $N \times N$, где каждый пиксель занимает 4 байта. Напишите метод, поворачивающий изображение на 90° .

7 Напишите алгоритм, реализующий следующее условие: если элемент матрицы в точке $M \times N$ равен 0, то весь столбец и вся строка обнуляются.

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

Дополнительные вопросы: манипуляция с битами (5.7), объектно-ориентированное проектирование (7.10), рекурсия (8.3), сортировка и поиск (9.6), C++ (13.10), моделирование (17.7, 17.8, 17.14).

2. Связные списки

Вопросы про связные списки могут показаться трудными. Но есть и хорошие новости — таких вопросов немного, а множество задач являются всего лишь разновидностями типовых вопросов о связных списках.

Задачи на связные списки предполагают, что вы можете реализовать связный список с нуля.

Создание связного списка

Следующий код реализует очень простой односвязный список:

```
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}
```

Обратите внимание: когда вы говорите о связном списке на собеседовании, вы должны понимать, о каком списке — односвязном или двусвязном — идет речь.

Удаление узла из односвязного списка

Удаление узла из односвязного списка — достаточно простая задача. Дан узел `n`, нам нужно найти предыдущий узел `prev` и установить `prev.next` в `n.next`. Если у нас есть двусвязный список, нам нужно обновить `n.next`, чтобы установить `n.next.prev` в `n.prev`. Не забудьте сделать проверку на `null`-указатель и при необходимости обновить начало и конец списка.

Если вы реализуете код на C/C++ или другом языке, требующем контроля памяти, вам нужно 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; /* перемещаем голову */  
6     }  
7  
8     while (n.next != null) {
```

```

9     if (n.next.data == d) {
10         n.next = n.next.next;
11         return head; /* голова списка не изменяется */
12     }
13     n = n.next;
14 }
15 return head;
16 }
```

Метод бегунка

Метод бегунка (или второго указателя) используется во многих задачах по связанным спискам. Вы «пробегаетесь» по связному списку двумя указателями одновременно. При этом один указатель называется «быстрым», а другой — «медленным».

Лучше всего продемонстрировать этот метод на примере. Существует связный список $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_n$ и его необходимо преобразовать в вид: $a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2 \rightarrow \dots \rightarrow a_n \rightarrow b_n$. Вы не знаете длину связного списка, но вы точно знаете, что длина — четное число.

У вас есть два указателя — $p1$ (быстрый) и $p2$ (медленный). За одну итерацию $p1$ перемещается на два элемента, а $p2$ — на один. 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

Много задач о связных списка решаются с помощью рекурсии. Получив задание, вы должны проверить, можно ли использовать рекурсивный подход. Сейчас мы не будем подробно говорить о рекурсии, этой теме будет посвящен отдельный раздел.

Вы должны помнить, что рекурсивные алгоритмы занимают как минимум $O(n)$ пространства, где n — глубина рекурсии. Все рекурсивные алгоритмы можно заменить на итерационные, но более сложные алгоритмы.

Вопросы собеседования

- 2.1. Напишите код, удаляющий дубликаты из несортированного связного списка.
Дополнительно
Как вы будете решать задачу, если запрещается использовать временный буфер?
- 2.2. Реализуйте алгоритм для поиска в односвязном списке k-го элемента с конца.
- 2.3. Реализуйте алгоритм, удаляющий узел из середины односвязного списка (доступ дан только к этому узлу).

Пример

Ввод: узел с из списка $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

Вывод: ничего не возвращается, но новый список имеет вид: $a \rightarrow b \rightarrow d \rightarrow e$

- 2.4.** Напишите код, разбивающий связный список вокруг значения x , так чтобы все узлы, меньшие x , оказались перед узлами, большими или равными x .
- 2.5.** Два числа хранятся в виде связных списков, в которых каждый узел содержит один разряд. Все цифры хранятся в обратном порядке, при этом первая цифра числа находится в начале списка. Напишите функцию, которая суммирует два числа и возвращает результат в виде связного списка.

Пример

Ввод: $(7 \rightarrow 1 \rightarrow 6) + (5 \rightarrow 9 \rightarrow 2)$. Это означает $617 + 295$.

Выход: $2 \rightarrow 1 \rightarrow 9$, что означает 912 .

Дополнительно

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

Ввод: $(6 \rightarrow 1 \rightarrow 7) + (2 \rightarrow 9 \rightarrow 5)$. Это означает $617 + 295$.

Выход: $9 \rightarrow 1 \rightarrow 2$, что означает 912 .

- 2.6.** Для кольцевого связного списка реализуйте алгоритм, возвращающий начальный узел петли.

Определение

Кольцевой связный список – это связный список, в котором указатель последующего узла связан с более ранним узлом, образуя петлю.

Пример

Ввод: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow C$ (предыдущий узел C)

Выход: C

- 2.7.** Реализуйте функцию, проверяющую, является ли связный список палиндромом.

Дополнительные вопросы: деревья и графы (4.4), объектно-ориентированное проектирование (7.10), масштабируемость и лимиты памяти (11.7), моделирование (17.13).

Удаление узла из односвязного списка

Удаление узла из односвязного списка – это операция, при которой из связного списка исключается определенный узел. Для этого необходимо помнить, что у каждого узла есть указатель на следующий узел. Но забудьте о том, что у последнего узла нет указателя на следующий узел, это называется концом списка.

Существует несколько способов удаления узла из списка. Один из них – это перебор всех узлов списка и поиск нужного узла. Другой способ – это использование дополнительного узла, который будет содержать информацию о том, какой узел нужно удалить. Третий способ – это использование двусвязного списка, где каждый узел имеет указатели как на предыдущий, так и на следующий узел.

Еще один способ – это использование стека. Стек – это структура данных, которая работает по принципу «последний вошел – первый вышел». Для удаления узла из списка, нужно сначала把他存入栈中, 然后从栈顶取出该节点, 并将其指针指向其前一个节点。如果当前节点是链表的头节点, 那么需要重新设置头节点为当前节点的下一个节点。

3. Стек и очередь

Как и в случае связных списков, на вопросы по стекам и очередям проще отвечать, когда известны входные и выходные структуры данных. Некоторые задачи являются небольшими модификациями исходной структуры данных, но вам может достаться и более сложное задание.

Реализация стека

Стек использует порядок LIFO (последним вошел, первым вышел). Стек подобен стопке тарелок — последнюю добавленную в стопку тарелку возьмут первой.

Давайте рассмотрим простой код для демонстрации работы стека. Стек можно реализовать с помощью связного списка. Фактически, стек — это связный список, не позволяющий пользователю получить доступ к элементам ниже главного узла.

```
1 class Stack {  
2     Node top;  
3  
4     Object pop() {  
5         if (top != null) {  
6             Node item = top.data;  
7             top = top.next;  
8             return item;  
9         }  
10        return null;  
11    }  
12  
13    void push(Object item) {  
14        Node t = new Node(item);  
15        t.next = top;  
16        top = t;  
17    }  
18  
19    Object peek() {  
20        return top.data;  
21    }  
22 }
```

Реализация очереди

Очередь использует порядок FIFO (первым вошел, первым вышел). Элементы удаляются из очереди в том же порядке, в котором были добавлены.

Очередь может также быть реализована в виде связного списка с добавлением новых пунктов в его конец.

```
1 class Queue {  
2     Node first, last;  
3  
4     void enqueue(Object item) {  
5         Node n = new Node(item);  
6         if (first == null) {  
7             first = n;  
8             last = n;  
9         } else {  
10            last.next = n;  
11            last = n;  
12        }  
13    }  
14  
15    Object dequeue() {  
16        if (first == null) {  
17            return null;  
18        } else {  
19            Object item = first.data;  
20            first = first.next;  
21            if (first == null) {  
22                last = null;  
23            }  
24            return item;  
25        }  
26    }  
27}
```

продолжение ↗

```

5     if (!first) {
6         last = new Node(item);
7         first = last;
8     } else {
9         last.next = new Node(item);
10        last = last.next;
11    }
12 }
13
14 Node dequeue(Node n) {
15     if (first != null) {
16         Object item = first.data;
17         first = first.next;
18         return item;
19     }
20     return null;
21 }
22 }
```

Вопросы собеседования

- 3.1. Опишите, как можно использовать один одномерный массив для реализации трех стеков.
- 3.2. Как реализовать стек, в котором кроме стандартных функций `push` и `pop` будет использоваться функция `min`, возвращающая минимальный элемент? Оценка времени работы функций `push`, `pop` и `min` — $O(1)$.
- 3.3. Представьте стопку тарелок. Если стопка слишком высокая, она может развалиться. В реальной жизни, когда высота стопки превысила бы некоторое 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 работать с общим стеком (со всей структурой) и должны возвращать те же значения, как если бы у нас был один большой стек.

Дополнительно

Реализуйте функцию `popAt(int index)`, которая осуществляет операцию `pop` в указанный под-стек.

- 3.4. В задаче про Ханойскую башню задействованы 3 башни и N дисков разных размеров, которые нужно переместить, (больший диск нельзя класть на меньший). Имеются следующие ограничения:
 - 1) за один раз можно переместить только один диск;
 - 2) диски перемещаются только с вершины одной башни на другую башню;
 - 3) диск можно положить только поверх большего диска.
 Напишите программу перемещения дисков (с первой башни на последнюю) с использованием стеков.

-
- 3.5. Создайте класс `MyQueue`, который реализует очередь с использованием двух стеков.
-
- 3.6. Напишите программу сортировки стека по возрастанию. Можно использовать дополнительные стеки для хранения элементов, но нельзя копировать элементы в другие структуры данных (например, в массив). Стек поддерживает следующие операции: `push`, `pop`, `peek`, `isEmpty`.
-
- 3.7. 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 выбирать кошку или собаку (животное в любом случае будет «самым старым»). Нельзя выбрать любое понравившееся животное. Создайте структуру данных, обслуживающую эту систему и реализующую операции `enqueue`, `dequeueAny`, `dequeueDog` и `dequeueCat`. Вы должны использовать встроенную структуру данных `LinkedList`.
-

Дополнительные вопросы: связные списки (2.7), математика и вероятность (10.7).

4. Деревья и графы

Очень часто кандидаты уверены, что задачи о деревьях и графах содержат больше всего подвохов. Вариант поиска структуры данных сложнее, чем вариант с линейно-организованной структурой (массив или связный список). Кроме того, время выполнения в наихудшем случае и среднее время могут существенно различаться, а вы должны оценить оба аспекта любого алгоритма. Скорость, с которой вы реализуете дерево или граф «с нуля», также играет важную роль.

Потенциальные ловушки

Деревья и графы — рассадник неоднозначностей и неправильных предположений. Убедитесь, что не упустили из виду какие-либо аспекты и сможете обосновать ваше решение в случае необходимости.

Бинарное дерево vs бинарное дерево поиска

Когда задан 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 дети меньше или равны текущему узлу, а правые — больше текущего узла.

Сбалансировано vs несбалансировано

В большинстве случаев предполагается, что деревья сбалансированы, но не всегда. Попросите интервьюера уточнить задачу. Если дерево несбалансировано, вам придется создавать алгоритм, учитывая оптимизацию. Существует множество способов сбалансировать дерево так, чтобы глубина поддеревьев оставалась в пределах заданного диапазона. Но это не означает, что левое и правое поддеревья будут одинакового размера.

Полнота дерева

Полные деревья — это деревья, в которых все листья находятся внизу, а все остальные узлы, не являющиеся листьями, имеют по два потомка. Нужно отметить, что полные деревья — чрезвычайная редкость, так как дерево должно содержать 2^n узлов, чтобы удовлетворять этому условию.

Обход бинарного дерева

Перед собеседованием попробуйте разобраться в реализации обхода дерева в прямом и обратном порядке. Наиболее часто используется обход в прямом порядке, левое поддерево — вершина — правое поддерево.

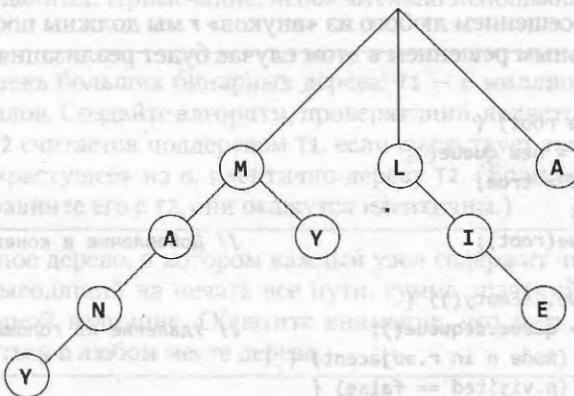
Балансировка: красно-черные и АВЛ-деревья

Хотя знание способов балансировки дерева поможет продемонстрировать, что вы являетесь великолепным разработчиком программного обеспечения, эти вопросы очень редко возникают во время собеседования. Вы должны знать время выполнения

операций на сбалансированных деревьях и быть знакомы с основными способами балансировки, но подробности мало кого заинтересуют.

Предиксное дерево

Предиксное (нагруженное) дерево (trie) — это разновидность обычного n -арного дерева, в узлах которого хранятся не ключи, а символьные метки. Каждый путь по такому дереву является словом. Простое предиксное дерево имеет следующий вид:



Обход графа

Большинство кандидатов неплохо разбираются в бинарных деревьях, но обход графа требует несколько иного подхода. Особенно сложным является поиск в ширину. Не забудьте, что поиск в ширину (BFS, Breadth First Search) и поиск в глубину (DFS, Depth First Search) предназначены для разных сценариев. DFS — это самый простой способ посетить все узлы в графе, или, по крайней мере, посетить каждый узел, пока не найдется искомый. Однако при работе с очень большими деревьями DFS не подходит. В этих случаях лучше воспользоваться BFS.

Поиск в глубину

При DFS мы посещаем узел r и затем проходимся по всем ближайшим к r узлам. При попадании в узел n (соседний с r) мы посещаем всех его соседей. Таким образом, перед перемещением к следующему потомку узла r полностью обследуется узел n .

Обратите внимание, что прямой порядок и другие формы обхода дерева являются разновидностями DFS. Основное отличие состоит в том, что при реализации алгоритма в случае графа необходимо проверить, был ли посещен узел. Если этого не сделать, то можно застрять в бесконечном цикле.

Приведенный псевдокод реализует DFS:

```

1 void search(Node root) {
2     if (root == null) return;
3     visit(root);
4     root.visited = true;
5     foreach (Node n in root.adjacent) {
  
```

продолжение ↗

```

6     if (n.visited == false)
7         search(n);
8     }
9 }
10 }

```

Поиск в ширину

Поиск в ширину (BFS) не так прост, у большинства кандидатов при первом знакомстве он вызывает затруднения.

При BFS перед посещением любого из «внуков» r мы должны посетить всех соседей с узла r . Оптимальным решением в этом случае будет реализация с циклом и очередью:

```

1 void search(Node root) {
2     Queue queue = new Queue();
3     root.visited = true;
4     visit(root);
5     queue.enqueue(root); // Добавление в конец очереди
6
7     while (!queue.isEmpty()) {
8         Node r = queue.dequeue(); // Удаление из головы очереди
9         foreach (Node n in r.adjacent) {
10             if (n.visited == false) {
11                 visit(n);
12                 n.visited = true;
13                 queue.enqueue(n);
14             }
15         }
16     }
17 }

```

Если вас попросят реализовать BFS, вспомните, что ключевым моментом является использование очереди. Остальную часть алгоритма можно построить исходя из этого факта.

Вопросы собеседования

- 4.1.** Реализуйте функцию, проверяющую сбалансированность бинарного дерева. Предположим, что дерево считается сбалансированным, если разница высот двух поддеревьев любого узла не превышает 1.
- 4.2.** Разработайте алгоритм поиска маршрута между двумя узлами для направленного графа.
- 4.3.** Напишите алгоритм создания бинарного дерева поиска с минимальной высотой для отсортированного (по возрастанию) массива.
- 4.4.** Для бинарного дерева поиска разработайте алгоритм, создающий связный список, состоящий из всех узлов заданной глубины (для дерева с глубиной D должно получиться D связных списков).

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

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

4.7. Создайте алгоритм и напишите код поиска первого общего предка двух узлов бинарного дерева. Постарайтесь избежать хранения дополнительных узлов в структуре данных. Примечание: необязательно использовать бинарное дерево поиска.

4.8. Дано: два очень больших бинарных дерева: T_1 – с миллионами узлов и T_2 – с сотнями узлов. Создайте алгоритм, проверяющий, является ли T_2 поддеревом T_1 . Дерево T_2 считается поддеревом T_1 , если существует такой узел n в T_1 , что поддерево, «растущее» из n , идентично дереву T_2 . (Если вы вырежете дерево в узле n и сравните его с T_2 , они окажутся идентичны.)

4.9. Дано бинарное дерево, в котором каждый узел содержит число. Разработайте алгоритм, выводящий на печать все пути, сумма значений которых соответствует заданной величине. Обратите внимание, что путь может начинаться и заканчиваться в любом месте дерева.

Дополнительные вопросы: сортировка и поиск (9.8), масштабируемость и лимиты памяти (11.2, 11.5), модерирование (17.13, 17.14), тяжелые вычисления (18.6, 18.8, 18.9, 18.10, 18.13).

5. Поразрядная обработка

Поразрядная обработка используется при решении самых разных задач. Иногда в задании явно указывается, что нужно произвести побитовую обработку, но, скорее всего, она вам понадобится для оптимизации кода. Вы должны ориентироваться в этой методике, как при вычислениях, так и при написании кода. Но будьте осторожны: при битовой обработке очень легко совершить ошибку. Обязательно протестируйте получившийся код или проверяйте его непосредственно во время написания.

Расчеты на бумаге

Ниже приведены практические задания, которые помогут вам преодолеть боязнь поразрядной обработки. Если вы «застряли», взгляните на эти операции как на операции с обычными числами в десятичной системе счисления, а затем примените этот же подход к бинарной арифметике.

Не забудьте некоторые обозначения: \wedge соответствует операции XOR, а \sim — операции NOT. Предположим, что мы работаем с четырехбитными числами. Задания из третьей колонки можно решить или использовать приведенные ниже «хитрости».

$0110 + 0010$	$0011 * 0101$	$0110 + 0110$
$0011 + 0010$	$0011 * 0011$	$0100 * 0011$
$0110 - 0011$	$1101 \gg 2$	$1101 \wedge (\sim 1101)$
$1000 - 0110$	$1101 \wedge 0101$	$1011 \& (\sim 0 \ll 2)$

Решение: строка 1 ($1000, 1111, 1100$); строка 2 ($0101, 1001, 1100$); строка 3 ($0011, 0011, 1111$); строка 4 ($0010, 1000, 1000$).

Трюки для третьей колонки:

- $0110 + 0110 = 0110 * 2$, что эквивалентно смещению 0110 влево на 1.
- Поскольку $0100 = 4$, можно умножить 0011 на 4. Умножение на 2^n сдвигает разряды числа на n . Мы смещаем 0011 влево на 2 и получаем 1100.
- Взгляните на эту операцию с точки зрения битов. Операция XOR для бита и его инверсии всегда дает 1. Поэтому $a \wedge (\sim a)$ всегда дает последовательность единиц.
- Операция вида $x \&& (\sim 0 \ll n)$ очищает n правых битов числа x . Значение ~ 0 — это последовательность единиц. При сдвиге числа влево на n позиций мы получаем последовательность единиц, за которыми следуют n нулей. Операция AND полученного числа с числом x очищает правые n битов числа x .

Чтобы решить остальные задачи, воспользуйтесь калькулятором Windows в режиме Вид → Программист. Так вам будет проще выполнять бинарные операции, включая AND, XOR и сдвиги.

Биты: трюки и факты

При решении задач на битовые операции полезно знать некоторые факты. Их нужно запомнить, а понять. Записи **1s** и **0s** используются для обозначения последовательностей единиц или нулей соответственно.

$$\begin{array}{lll} x \wedge 0s = x & x \wedge 1s = 0 & x \mid 0s = x \\ x \wedge 1s = \sim x & x \wedge 1s = x & x \mid 1s = 1s \\ x \wedge x = 0 & x \wedge x = x & x \mid x = x \end{array}$$

Основные задачи: получение, установка, очистка и обновление бита

Приведенные далее операции очень важны, вам нужно понимать, как они работают. Зубрежка обязательно приведет к ошибкам. Вместо этого разберитесь, как можно реализовать эти методы, тогда вы сможете решить любые задачи, связанные с битами.

Извлечение бита

Метод `getBit` сдвигает 1 на *i* битов, создавая значение, например **00010000**. Операция `AND` над числом `num` очищает все биты, кроме бита *i*. Затем этот бит сравнивается с 0. Если результат не равен 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 }
```

Очистка бита

Этот метод — противоположность метода `setBit`. Сначала создается число **11101111** (инверсия числа **00010000**). Затем производится операция `AND` с числом `num`. Таким образом, очищается только 1-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 }
```

Для очистки всех битов до бита *i* (включительно):

```
1 int clearBitsMSBthroughI(int num, int i) {  
2     int mask = (1 << (i+1)) - 1;  
3     return num & mask;  
4 }
```

Для очистки всех битов от i до 0 (включительно):

```
1 public static int clearBitsIthrough0(int num, int i) {
2     int mask = ~((1 << (i+1)) - 1);
3     return num & mask;
4 }
```

Обновление бита

Этот метод является объединением методов `setBit` и `clearBit`. Сначала с помощью маски, например `11101111`, очищается бит на позиции i . Затем значение v сдвигается вправо на i битов. В результате создается число, у которого i -й бит равен v , а все остальные биты нулевые. Операция `OR` для этих 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, int v) {
2     int mask = ~(1 << i);
3     return (num & mask) | (v << i);
4 }
```

Вопросы собеседования

- 5.1.** Дано: два 32-битных 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 . Можно считать, что j и i имеют такие значения, что число M обязательно поместится в этот промежуток. Если $M = 10011$, для его размещения понадобится 5 битов между j и i . Если $j = 3$ и $i = 2$, число M не поместится в указанный промежуток.

Пример

Ввод: $N = 10000000000$, $M = 10011$, $i = 2$, $j = 6$

Выход: $N = 10001001100$

- 5.2.** Дано: вещественное число в интервале между 0 и 1 (например, 0,72), которое было передано как `double`. Запишите его двоичное представление. Если для представления числа не хватает 32 разрядов, выведите сообщение об ошибке.

- 5.3.** Дано: положительное число. Выведите ближайшие наименьшее и наибольшее числа, которые имеют такое же количество единичных битов в двоичном представлении.

- 5.4.** Объясните, что делает код: `((n & (n-1)) == 0)`.

- 5.5.** Напишите функцию, определяющую количество битов, которые необходимо изменить, чтобы из целого числа A получить целое число B .

Пример

Ввод: 31, 14

Выход: 2

- 5.6.** Напишите программу, меняющую местами четные и нечетные биты числа. Количество инструкций должно быть наименьшим (нужно поменять местами биты 0 и 1, 2 и 3 и т. д.).

- 5.7. Массив $A[1\dots n]$ содержит целые числа от 0 до n , но одно число отсутствует. В этой задаче мы не можем получить доступ к любому числу в массиве A с помощью одной операции. Элементы массива A хранятся в двоичном виде, а доступ к ним осуществляется только при помощи команды извлечь j -й бит из $A[i]$, имеющей фиксированное время выполнения. Напишите код, обнаруживающий отсутствующее целое число. Можно ли выполнить эту задачу за время $O(n)$?
- 5.8. Изображение с монохромного экрана сохранено как одномерный массив байтов, так что в одном байте хранится информация о восьми соседних пикселях. Ширина изображения w кратна 8 (байты соответствуют столбцам). Высоту экрана можно рассчитать, зная длину массива и ширину экрана. Реализуйте функцию `drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)`, которая рисует горизонтальную линию из точки $(x1, y)$ в точку $(x2, y)$.

Дополнительные вопросы: массивы и строки (1.1, 1.7), рекурсия (8.4, 8.11), масштабируемость и лимиты памяти (11.3, 11.4), C++ (13.9), моделирование (17.1, 17.4), задачи повышенной сложности (#18.1).

Использование

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 единого подхода к тому, какую задачу можно отнести к головоломкам, не существует.

Если вам досталась головоломка, то наверняка она интересная и наверняка ее можно решить логически. Корни большинства головоломок лежат в математике или информатике.

Давайте рассмотрим основные подходы к решению головоломок.

Начните говорить

Если вам дали головоломку, не паникуйте. Как и в случае с задачами на алгоритмы, интервьюеры хотят видеть, что 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 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. 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.

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

Большинство головоломок относятся к задачам минимизации, когда требуется уменьшить количество действий или выполнить что-либо определенное количество раз. Можно попробовать «сбалансировать» худший случай. Таким образом, если решение приведет к смещению худшего случая, можно выполнить балансировку худшего случая. Рассмотрим конкретный пример.

Задача о «девяти шарах» — классика собеседования. У вас есть 9 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!

Это пример несбалансированного худшего случая. Чтобы определить, является ли девятый шар самым тяжелым, нужно одно взвешивание, а чтобы выяснить, где он, — три. Если мы «оштрафуем» девятый шар, отложив большее количество шаров, то можем снизить нагрузку на другие. Классический пример балансировки худшего случая.

Если мы раздели 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: дано 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. Если шары весят одинаково, значит, оставшийся шар — самый тяжелый, иначе весы «покажут» тяжелый шар.

Алгоритмический подход

Если вы не можете сразу найти решение, попробуйте использовать один из методов алгоритмизации. Головоломки — это те же задачи алгоритмизации, из которых

убраны технические нюансы. Думайте, упрощайте, обобщайте, сопоставляйте с шаблоном, используйте базовый случай — все это может пригодиться.

Вопросы собеседования

- 6.1. Дано: 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?
- 6.2. Дано: 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 свой ответ.
- 6.3. 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.
- 6.4. На острове существует правило — голубоглазые люди не могут там находиться. Самолет улетает каждый вечер в 20:00. Каждый человек может видеть цвет глаз других людей, но не знает цвет собственных, никто не имеет права сказать человеку, какой у него цвет глаз. На острове находится не менее одного голубоглазого человека. Сколько дней потребуется, чтобы все голубоглазые уехали?
- 6.5. Дано 100-этажное здание. Если яйцо сбросить с высоты N -го этажа (или с большей высоты), оно разбьется. Если его бросить с меньшего этажа, оно не разбьется. У вас есть два яйца, найдите N за минимальное количество бросков.
- 6.6. Дано: 100 закрытых замков расположены в длинном коридоре. 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). На 100-м проходе человек должен «переключить» только замок № 100. Сколько замков остались открытыми?

7. Математика и теория вероятностей

Хотя значительное количество математических задач, которые приходится решать во время собеседования, похожи на головоломки, для их решения можно использовать логический подход. Они базируются на основных законах математики и логики, и знание этого факта облегчит как само решение, так и его проверку. Сейчас мы поговорим об основных математических понятиях, которые могут вам пригодиться.

Простые числа

Любое число может быть представлено как произведение простых чисел. Например:

$$84 = 2^2 \times 3^1 \times 5^0 \times 7^1 \times 11^0 \times 13^0 \times 17^0 \times \dots$$

Делимость

Из закона простых чисел следует, что если x делится на y (будем записывать это условие как $x \mid y$ или $\text{mod}(y, x) = 0$), то все простые числа, входящие в разложение на множители числа x , должны присутствовать в разложении на множители числа y . Рассмотрим пример:

Пусть $x = 2^{j_0} * 3^{j_1} * 5^{j_2} * 7^{j_3} * 11^{j_4} * \dots$

Пусть $y = 2^{k_0} * 3^{k_1} * 5^{k_2} * 7^{k_3} * 11^{k_4} * \dots$

Если $x \mid y$, тогда для всех i выполняется неравенство $j_i \leq k_i$.

Наибольший общий делитель x и y :

$$\text{gcd}(x, y) = 2^{\min(j_0, k_0)} * 3^{\min(j_1, k_1)} * 5^{\min(j_2, k_2)} * \dots$$

Наименьшее общее кратное x и y :

$$\text{lcm}(x, y) = 2^{\max(j_0, k_0)} * 3^{\max(j_1, k_1)} * 5^{\max(j_2, k_2)} * \dots$$

Попробуйте сами решить забавную задачку: найдите произведение НОД и НОК ($\text{gcd} * \text{lcm}$):

$$\begin{aligned}\text{gcd} * \text{lcm} &= 2^{\min(j_0, k_0)} * 2^{\max(j_0, k_0)} * 3^{\min(j_1, k_1)} * 3^{\max(j_1, k_1)} * \dots \\ &= 2^{\min(j_0, k_0) + \max(j_0, k_0)} * 3^{\min(j_1, k_1) + \max(j_1, k_1)} * \dots \\ &= 2^{j_0 + k_0} * 3^{j_1 + k_1} * \dots \\ &= 2^j * 3^k * \dots \\ &= xy\end{aligned}$$

Является ли число простым

Этот вопрос настолько часто встречается, что о нем необходимо упомянуть. Для этого достаточно проверить делимость в цикле от 2 до $n-1$:

```
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 немаловажное улучшение — цикл можно ограничить квадратным корнем 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 }
```

Квадратного корня достаточно, поскольку для каждого числа, которое делится на n без остатка, есть дополнение b , такое что $a * b = n$. Если $a > \sqrt{n}$, то $b < \sqrt{n}$ (поскольку $\sqrt{n} * \sqrt{n} = n$). а нам не понадобится, так как для проверки n на простоту мы уже сверились с b .

На самом деле все, что нужно сделать, — это проверить, делится ли n на простые числа. И тут появляется решето Эратосфена.

Список простых чисел: решето Эратосфена

Решето Эратосфена — эффективный способ генерации списка простых чисел. Он работает, распознавая все составные числа, которые делятся на простые числа.

Начнем с генерации списка всех чисел, не превышающих заданного значения `max`. Можно сразу вычеркнуть четные числа. Затем найти ближайшее простое число и вычеркнуть все числа, кратные ему. Последовательно исключая все числа, кратные 2, 3, 5, 7, 11 и т. д., мы получим список простых чисел, лежащих в диапазоне от 2 до `max`.

Представленный далее код реализует решето Эратосфена:

```

1 boolean[] sieveOfEratosthenes(int max) {
2     boolean[] flags = new boolean[max + 1];
3     int count = 0;
4
5     init(flags); // Устанавливаем все флаги в true (кроме 0 и 1)
6     int prime = 2;
7
8     while (prime <= max) {
9         /* Вычеркиваем оставшуюся часть произведений простых чисел */
10        crossOff(flags, prime);
11
12        /* Находим следующее истинное значение */
13        prime = getNextPrime(flags, prime);
14
15        if (prime >= flags.length) {
16            break;
17        }
18    }
19
20    return flags;
21 }
22 }
```

```

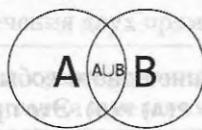
23 void crossOff(boolean[] flags, int prime) {
24     /* Вычеркиваем оставшуюся часть произведений простых чисел. Мы начинаем
25     * с (prime*prime), потому что, если существует k * prime,
26     * где k < prime, это значение должно быть вычеркнуто
27     * на предыдущей итерации. */
28     for (int i = prime * prime; i < flags.length; i += prime) { (A)q = (B по A)q
29         flags[i] = false;                                         в итоге мы имеем
30     }                                                       итогово значение в
31 }                                                       итогово значение в
32
33 int getNextPrime(boolean[] flags, int prime) {
34     int next = prime + 1;
35     while (next < flags.length && !flags[next]) { в итоге. Существует два вариан-
36         next++;                                         та для каждого из них
37     }
38     return next;                                         в итоге. Существует два вариан-
39 }

```

Конечно, данную программу можно дополнительно оптимизировать. Простейший способ — использовать массив нечетных чисел, который сократит использование памяти в два раза.

Теория вероятностей

Теория вероятностей — довольно сложная тема, но она основана на нескольких достаточно логичных законах. Посмотрите на диаграмму Венна, визуализирующую два события — А и В. Области двух кругов соответствуют относительным вероятностям событий, а область пересечения — вероятности события $\{A \text{ and } B\}$.



Вероятность события $\{A \text{ and } B\}$

Представьте, что вы бросили стрелку дартса в диаграмму Венна. Какова вероятность, что вы попадете в пересечение кругов А и В? Если вы знаете вероятность попадания в область А и знаете, какой процент круга А перекрывается кругом В (то есть вероятность того, что вы попадете в В, если попали в А), то вероятность события $P\{A \text{ and } B\}$ можно записать как:

$$P(A \text{ and } B) = P(B \text{ given } A) \cdot P(A)$$

Например, предположим, что мы выбрали число из интервала от 1 до 10 (включительно). Какова вероятность того, что выбранное число окажется четным и будет принадлежать интервалу от 1 до 5. Вероятность того, что выбранное число принадлежит диапазону от 1 до 5, составляет 50 %, а вероятность того, что это число окажется четным, — 40 %. Так, подсчитаем вероятность данного события:

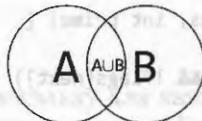
$$\begin{aligned}
 P(x \text{ is even and } x \leq 5) &= P(x \text{ is even given } x \leq 5) \cdot P(x \leq 5) \\
 &= (2/5) \cdot (1/2) \\
 &= 1/5
 \end{aligned}$$

Вероятность события {A or B}

Рассмотрим другой случай — вероятность попадания в область А или В — $P\{A \text{ or } B\}$. Если вы знаете вероятности попадания в каждую область и вероятность попадания в пересечение областей, тогда

$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

С точки зрения логики все корректно — нам нужно только сложить площади обоих кругов и избавиться от двойного наложения в области пересечения. Давайте визуализируем эту задачу с помощью диаграммы Венна:



Например, предположим, что мы выбрали число из интервала от 1 до 10 (включительно). Какова вероятность того, что число окажется четным **или** будет принадлежать диапазону от 1 до 5? Вероятность того, что число является четным, — 50 %. Вероятность того, что число принадлежит интервалу от 1 до 5, также составляет 50 %. Вероятность того, что число четное **и** принадлежит заданному интервалу, мы уже вычислили в предыдущем примере, она составляет 20 %.

$$\begin{aligned} P(x \text{ is even or } x \leq 5) &= P(x \text{ is even}) + P(x \leq 5) - P(x \text{ is even and } x \leq 5) \\ &= (1/2) + (1/2) - (1/5) \\ &= 4/5 \end{aligned}$$

Теперь сформулировать правила для независимых и взаимоисключающих событий не составит труда.

Независимость событий

Если А и В независимы (наступление одного события не изменяет вероятность наступления другого), то $P(A \text{ and } B) = P(A) P(B)$. Это правило с очевидностью следует из того, что $P(B \text{ given } A) = P(B)$, при условии, что событие А никак не связано с В.

Взаимоисключающие события

Если А и В являются взаимоисключающими (если одно событие произошло, то другое произойти не может), то $P(A \text{ or } B) = P(A) + P(B)$ (поскольку $P(A \text{ and } B) = 0$).

Многие путают независимые и взаимоисключающие события. На самом деле два события не могут быть одновременно независимыми и взаимоисключающими (конечно, если вероятность событий больше 0). Почему? Для взаимоисключающих событий факт свершения одного события означает, что второе невозможно. Независимость означает, что реализация одного события никак не влияет на другое. Таким образом, два события с ненулевыми вероятностями не могут быть и взаимоисключающими, и независимыми.

Если одно или оба события имеют нулевую вероятность (то есть события невозможны), то события являются и независимыми, и взаимоисключающими. Это следует из определений независимых и взаимоисключающих событий.

Обратите внимание!

Будьте осторожны, работая с типами `float` и `double`, — у них разное количество разрядов.

Не предполагайте, что речь идет о целых числах, если это не указано явно.

Не предполагайте, что события являются независимыми (или взаимоисключающими), если это явно не указано. Будьте внимательны, боритесь с искушением слепого умножения или сложения вероятностей.

Вопросы собеседования

- 7.1. Вам предлагают бросить мяч в баскетбольное кольцо. Существует два варианта игры:

Вариант 1: попасть в кольцо с одного броска.

Вариант 2: у вас есть три попытки, и нужно попасть в кольцо два раза из трех.

Если p — вероятность попадания, то как, в зависимости от значения p , выбрать более выигрышный вариант игры?

- 7.2. Три муравья находятся в вершинах треугольника. Какова вероятность столкновения между какими-либо двумя или всеми тремя муравьями, если муравьи начинают двигаться вдоль сторон треугольника? (Муравей выбирает любое направление с равной вероятностью, скорость движения муравьев одинакова.)

Найдите вероятность столкновения для случая, когда n муравьев находятся в вершинах n -угольника.

- 7.3. Какова вероятность пересечения двух прямых, лежащих в одной плоскости?

- 7.4. Используя только оператор суммирования, напишите методы, реализующие операции умножения, вычитания и деления целых чисел.

- 7.5. Для двух квадратов, лежащих в одной плоскости, найдите прямую, которая делила бы эти квадраты пополам. (Стороны квадратов параллельны осям координат.)

- 7.6. Дано: набор точек на плоскости. Найдите прямую линию, которая проходит через большинство точек.

- 7.7. Разработайте алгоритм, позволяющий найти k -е число из упорядоченного числового ряда, в разложении элементов которого на простые множители присутствуют только числа 3, 5 и 7.

Дополнительные вопросы: модерирование (17.11), задания повышенной сложности (18.2).

8. Объектно-ориентированное проектирование

Задания по объектно-ориентированному проектированию (ООП) предполагают, что кандидат может составить схему из классов и методов, позволяющую реализовать техническую задачу или имитировать реальный объект. Эти задачи дают интервьюеру возможность составить представление о стиле программирования кандидата.

Вы должны показать, что понимаете, как создать изящный и удобный объектно-ориентированный код. Так что провал при выполнении этих заданий может превратиться в «красную карточку» для всего собеседования.

Как подготовиться к заданиям по ООП

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, будете ли вы высказывать предположения или будете задавать уточняющие вопросы. В конце концов, разработчик, который пишет программы, должен сознавать, что его действия могут не только привести к потере времени и денег компании, но и создать более серьезные проблемы.

Задавая уточняющие вопросы по задачам ООП, вы выясняете, кто и как будет использовать ваш программный продукт. В этом диалоге вам следует руководствоваться правилом 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, целевая аудитория и условия использования устройства влияют на ваш проект.

Шаг 2. Определите основные объекты

Теперь, когда мы понимаем, что будем проектировать, 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. Анализируем 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? Какая связь используется — «многие-ко-многим» или «один-ко-многим»?

В случае с ресторанной кофеваркой, проект будет иметь вид:

- в **Party** присутствует массив **Guests**;
- **Server** и **Hosts** наследуются от **Employee**;
- у каждого **Table** есть один **Party**, но у каждого **Party** может быть несколько **Table**;
- один **Host** для **Restaurant**.

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 схема объектно-ориентированного проекта. Остается рассмотреть основные действия, которые могут быть выполнены объектами, и их взаимосвязи. Вы можете обнаружить, что забыли какие-либо объекты, и вам придется обновить проект.

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, компания (**Party**) занимает 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.

Разработка шаблонов

Поскольку интервьюеры пытаются протестировать ваши возможности, а не ваши знания, разработка шаблонов чаще всего остается за рамками собеседования. Однако шаблоны **Singleton** (одиночка) и **Factory Method** (фабричный метод) могут пригодиться, и мы остановимся на них поподробнее.

Существует множество шаблонов, но в этой книге мы не можем рассмотреть их все. Так что, если вы хотите улучшить навыки программирования, постарайтесь раздобыть книгу по этой теме.

Singleton

Шаблон **Singleton** гарантирует, что класс имеет только один экземпляр и обеспечивает доступ к экземпляру через приложение. Он может быть полезен в том случае, когда у вас есть «глобальный» объект с одним экземпляром. Например, мы можем реализовать класс **Restaurant** так, чтобы существовал единственный экземпляр **Restaurant**.

```
1 public class Restaurant {  
2     private Restaurant _instance = null;  
3     public static Restaurant getInstance() {  
4         if (_instance == null) {  
5             _instance = new Restaurant();  
6         }  
7         return _instance; // возвращаем один экземпляр  
8     }  
9 }
```

Factory Method

Фабричный метод предлагает интерфейс, позволяющий подклассам создавать экземпляры некоторого класса. Вам может понадобиться реализовать эту задачу с классом `Creator`, который является абстрактным, не способным обеспечить реализацию фабричного метода. Или же класс `Creator` может оказаться реальным классом, обеспечивающим реализацию этого метода. Тогда фабричный метод будет принимать параметр, определяющий, какой класс нужно инициализировать.

```

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

- 8.1. Разработайте структуры данных для универсальной колоды карт. Объясните, как разделить структуры данных на подклассы, чтобы реализовать игру в блэкджек.
- 8.2. Предположим, что существует Call-центр 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, звонок перенаправляется директору. Разработайте классы и структуры данных для этой задачи. Реализуйте метод `dispatchCall()`, который перенаправляет звонок первому свободному сотруднику.
- 8.3. Разработайте музыкальный автомат, используя принципы ООП.
- 8.4. Разработайте паркинг, используя принципы ООП.
- 8.5. Разработайте структуры данных для онлайн-библиотеки.
- 8.6. Запрограммируйте игру в пазл. Разработайте структуры данных и объясните алгоритм, позволяющий решить задачу. Вы можете предположить, что существует метод `fitsWith`, возвращающий значение `true` в том случае, если два переданных кусочка пазла должны располагаться рядом.
- 8.7. Как вы будете разрабатывать чат-сервер? Предоставьте информацию о компонентах бэкэнда, классах и методах. Перечислите самые трудные задачи, которые необходимо решить.
- 8.8. Правила игры «реверси» следующие. 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), его цвет меняется на

противоположный. Цель — захватить по крайней мере одну из фишечек противника. Игра заканчивается, когда у игрока не остается ходов. Побеждает тот, у которого больше фишечек на поле. Реализуйте ООП-модель для этой игры.

- 8.9.** Объясните, какие структуры данных и алгоритмы необходимо использовать для разработки файловой системы, хранящейся в оперативной памяти. Напишите программный код, иллюстрирующий использование этих алгоритмов.
- 8.10.** Спроектируйте и реализуйте хэш-таблицу, использующую связные списки для обработки коллизий.

Дополнительные вопросы: потоки и блокировки (16.3).

Время, потребованное на вычисление N-го числа Фибоначчи в секундах.

Очевидно, что время вычисления N-го числа Фибоначчи определяется тем количеством времени, затраченного на вычисление предыдущих чисел. Для этого можно использовать рекурсию. Рассмотрим, как это сделать.

Следующий код вычисляет N-ое число Фибоначчи с помощью рекурсии:

```
float fib (int n) {
    if (n <= 1) return 1.0;
    else return fib(n - 1) + fib(n - 2);
}
```

Когда мы хотим вычислить, скажем, 50-е число Фибоначчи, то в результате выполнения этого кода мы получим 12586269025. Но это не то, что мы хотим. Дело в том, что для вычисления 50-го числа Фибоначчи потребуется 50 вызовов функции. Каждый вызов занимает некоторое время. Метод динамического программирования позволяет сократить количество вызовов до 50.

Для этого мы можем запомнить результат каждого вычисления. Для этого мы можем создать массив `fib[51]`, в котором `fib[0] = 1`, `fib[1] = 1`. Каждый раз, когда нам потребуется вычислить, скажем, 50-е число Фибоначчи, мы сначала проверим, есть ли в массиве значение для этого числа. Если нет, то вычислим его и запомним в массиве.

Таким образом, мы можем избежать избыточных вычислений. Для этого мы можем создать массив `fib[51]`, в котором `fib[0] = 1`, `fib[1] = 1`. Каждый раз, когда нам потребуется вычислить, скажем, 50-е число Фибоначчи, мы сначала проверим, есть ли в массиве значение для этого числа. Если нет, то вычислим его и запомним в массиве.

Таким образом, мы можем избежать избыточных вычислений. Для этого мы можем создать массив `fib[51]`, в котором `fib[0] = 1`, `fib[1] = 1`. Каждый раз, когда нам потребуется вычислить, скажем, 50-е число Фибоначчи, мы сначала проверим, есть ли в массиве значение для этого числа. Если нет, то вычислим его и запомним в массиве.

9. Рекурсия и динамическое программирование

Существует огромное множество разнообразных рекурсивных задач, но большинство из них укладываются в определенные шаблоны. Подсказка — если задача рекурсивна, то она может быть разбита на подзадачи.

Когда вы получаете задание «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.

Практика — путь к совершенству! Чем больше задач вы решите, тем легче будет распознавать рекурсивные задачи.

С чего начать

Рекурсивные решения, по определению, основываются на решении подзадач. Очень часто придется вычислять $f(n)$, добавляя, вычитая или еще как-либо изменяя $f(n-1)$. Следует принимать во внимание оба типа рекурсии: восходящую и нисходящую.

Восходящая рекурсия

Восходящая рекурсия обычно более понятна. Мы знаем, как решить задачу для самого простого случая, например как вывести один элемент, затем решаем задачу для двух элементов, затем для трех и т. д. Подумайте о том, как создать решение для конкретного случая, основываясь на предыдущем решении.

Нисходящая рекурсия

Нисходящая рекурсия выглядит чуть более сложной, но она так же может понадобиться для решения задач. В этом случае мы должны решить, как разделить задачу на N подзадач. Не забывайте про перекрывающиеся случаи.

Динамическое программирование

Динамическое программирование (ДП) не очень популярно на собеседованиях, поскольку задачи по этой теме слишком сложны для 45-минутного интервью. Даже если хорошо подготовленные кандидаты смогут их решить, эти задачи — не самый удачный метод оценки кандидата.

Если вам не повезло и вы получили задачу по ДП, подход будет аналогичен подходу к решению рекурсивной задачи. Различие заключается в том, что промежуточные результаты «кэшируются» для будущих вызовов.

Простой пример динамического программирования: числа Фибоначчи

В качестве очень простого примера динамического программирования рассмотрим программу, генерирующую N -е число Фибоначчи. 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? Вычисление N -го числа Фибоначчи зависит от предыдущих $n-1$ чисел. Но каждый вызов делает два рекурсивных вызова. Это означает, что время выполнения составит $O(2^n)$. График, приведенный ниже, показывает это экспоненциальное увеличение.



Время, потраченное на вычисление N -го числа Фибоначчи в секундах

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[] fib = new int[max];
2 int fibonacci(int i) {
3     if (i == 0) return 0;
4     if (i == 1) return 1;
5     if (fib[i] != 0) return fib[i]; // Возвращаем кэшированный результат.
6     fib[i] = fibonacci(i - 1) + fibonacci(i - 2); // Кэшируем результат
7     return fib[i];
8 }
```

Генерирование 50-го числа Фибоначчи на стандартном компьютере с помощью рекурсивной функции займет более одной минуты. Метод динамического программирования позволит сгенерировать 10 000-е число Фибоначчи за доли миллисекунды (но не забывайте, что размера типа `int` может оказаться недостаточно).

В динамическом программировании нет ничего сложного — это все та же рекурсия, но с кэшированием результатов. Так что реализуйте обычное рекурсивное решение, а потом добавьте кэширование.

Рекурсивные и итерационные решения

Рекурсивные алгоритмы очень требовательны к памяти. Каждый рекурсивный вызов добавляет новый слой в стек. Если ваш алгоритм использует $O(n)$ рекурсивных вызовов, то для его работы понадобится $O(n)$ памяти. Много!

Любой рекурсивный алгоритм можно превратить в итерационный, хотя от этого код становится сложнее. Прежде чем написать рекурсивный код, задайте себе вопрос, как его можно преобразовать в итерационный, и не забудьте поговорить на эту тему с интервьюером.

Вопросы собеседования

- 9.1. Ребенок поднимается по лестнице из n ступенек, он может переместиться на одну, две или три ступеньки за один шаг. Реализуйте метод, рассчитывающий количество возможных вариантов прохождения ребенка по лестнице.
-

- 9.2. Представьте робота, сидящего в левом верхнем углу сетки с координатами X, Y . Робот может перемещаться в двух направлениях: вправо и вниз. Сколько существует маршрутов, проходящих от точки $(0, 0)$ до точки (X, Y) .

Дополнительно

Предположите, что на сетке существуют области, которые робот не может пересекать. Разработайте алгоритм построения маршрута от левого верхнего до правого нижнего угла.

- 9.3. В массиве $A[0..n-1]$ задан «волшебный» индекс — $A[i]=i$. Учитывая, что массив отсортирован, напишите метод, чтобы найти этот «волшебный» индекс, если он существует в массиве A .

Дополнительно

Что произойдет, если значения окажутся одинаковыми?

- 9.4. Напишите метод, возвращающий все подмножества одного множества.
-

- 9.5. Напишите метод, возвращающий все перестановки строки.
-

- 9.6. Реализуйте алгоритм, выводящий все корректные (правильно открытые и закрытые) комбинации пар круглых скобок.

Пример:

Ввод: 3

Вывод: ((())), ((())(), (())()), ()((())), ()()()

- 9.7. Реализуйте функцию заливки краской, которая используется во многих графических редакторах. Данна плоскость (двумерный массив цветов), точка и цвет, которым нужно заполнить все окружающее пространство, окрашенное в другой цвет.
-

- 9.8. Дано неограниченное количество монет достоинством 25, 10, 5 и 1 цент. Напишите код, определяющий количество способов представления n центов.
-

- 9.9. Данна шахматная доска размером 8×8 . Напишите алгоритм, находящий все варианты расстановки восьми ферзей так, чтобы никакие две фигуры не попадали на одну горизонталь, вертикаль или диагональ. Учитываются не только главные, но и все остальные диагонали.
-

- 9.10. Дан штабель из n ящиков шириной w_i , высотой h_i и глубиной d_i . Ящики нельзя поворачивать, добавлять ящики можно только наверх штабеля. Каждый нижний ящик в стопке по высоте, ширине и глубине больше ящика, который находится на нем. Реализуйте метод, позволяющий построить самый высокий штабель (высота штабеля равна сумме высот всех ящиков).
-

- 9.11. Дано логическое выражение, построенное из символов `0`, `1`, `&`, `|` и `^`, и имеющее значение `result`. Напишите функцию, подсчитывающую количество вариантов логических выражений, дающих значение `result`.

Пример:

Выражение: `1^0|0|1`

Результат: `result = false (0)`

Вывод: 2 способа: `1^((0|0)|1)` и `1^((0|(0|1))`

Дополнительные вопросы: связные списки (2.2, 2.5, 2.7), стеки и очереди (3.3), деревья и графы (4.1, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9), головоломки (6.4), сортировка и поиск (9.5, 9.6, 9.7, 9.8), C++ (13.7), моделирование (17.13, 17.14), задачи повышенной сложности (18.4, 18.7, 18.12, 18.13).

15 helper[5] = array[1];

16 /* проходимся по вспомогательному массиву от 0 до 4, т.к. вспомогательный массив имеет длину 5, а текущий индекс 4

17 int helperRight = middle + 1;

18 int current = low;

19 /*

20 /* проходимся по вспомогательному массиву от 0 до 4, т.к. вспомогательный массив имеет длину 5, а текущий индекс 4

21 while (helperLeft <= middle && helperRight <= 5) {

22 if (helper[helperLeft] == helper[helperRight]) {

23 array[current] = helper[helperLeft];

24 helperLeft++;

25 helperRight++;

26 current++;

27 }

28 /* проходимся по вспомогательному массиву от 0 до 4, т.к. вспомогательный массив имеет длину 5, а текущий индекс 4

29 for (int i = 0; i <= 4; i++) {

30 if (array[i] != array[i + 1]) {

31 cout << " " << array[i] << " " << array[i + 1] << endl;

32 }

33 /*

34 /*

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, какой и когда следует применять.

Предположим, что вас попросили выполнить следующее задание: провести сортировку очень большого массива `Person`, упорядочив его элементы в соответствии со значением возраста человека (в возрастающем порядке).

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, мы понимаем, что нам идеально подходит блочная сортировка. Если блоки будут достаточно маленькими (например, 1 год), то время выполнения составит $O(n)$.

Общие алгоритмы сортировки

Изучение (или повторение) алгоритмов сортировки — отличный способ повысить свои шансы на собеседовании. Ниже приведено описание алгоритмов, наиболее популярных у интервьюеров.

Пузырьковая сортировка | Время выполнения в худшем и среднем случае: $O(n^2)$. Память: $O(1)$.

Обработка начинается с первых элементов массива, они меняются местами, если первое значение больше, чем второе. Затем мы переходим к следующей паре и так далее, пока массив не будет отсортирован.

Сортировка выбором | Время выполнения в худшем и среднем случае: $O(n^2)$. Память: $O(1)$.

Сортировка выбором является вариантом предыдущего алгоритма: 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. В итоге вы объединяете два массива состоящие из одного элемента.

Метод слияния копирует все элементы из целевого массива во вспомогательные, разделяя левую и правую половины (`helperLeft` и `helperRight`). Мы «двигаемся» по вспомогательному массиву `helper`, копируя наименьший элемент каждой половины в массив. В итоге мы копируем оставшиеся элементы в целевой массив.

```

1 void mergesort(int[] array, int low, int high) {
2     if (low < high) {
3         int middle = (low + high) / 2;
4         mergesort(array, low, middle);           // Сортируем левую половину
5         mergesort(array, middle + 1, high);      // Сортируем правую половину
6         merge(array, low, middle, high);        // Объединяем их
7     }
8 }
9
10 void merge(int[] array, int low, int middle, int high) {
11     int[] helper = new int[array.length];
12
13     /* Копируем обе половины во вспомогательный массив */
14     for (int i = low; i <= high; i++) {
15         helper[i] = array[i];
16     }
17
18     int helperLeft = low;
19     int helperRight = middle + 1;
20     int current = low;
21
22     /* Проходимся по вспомогательному массиву. Сравниваем левую и правую
23     * половины, копируем обратно наименьший элемент из двух половин
24     * в исходный массив. */
25     while (helperLeft <= middle && helperRight <= high) {
26         if (helper[helperLeft] <= helper[helperRight]) {
27             array[current] = helper[helperLeft];
28             helperLeft++;
29         } else {                                // Если правый элемент меньше, чем левый
30             array[current] = helper[helperRight];
31             helperRight++;
32         }
33         current++;
34     }
35
36     /* Копируем оставшуюся часть левой стороны массива
37     * в целевой массив */
38     int remaining = middle - helperLeft;
39     for (int i = 0; i <= remaining; i++) {
40         array[current + i] = helper[helperLeft + i];
41     }
42 }

```

Обратите внимание, что в целевой массив скопированы только оставшиеся элементы левой половины вспомогательного массива. Почему не правой половины? Правая половина и не должна копироваться, потому что она уже там.

Рассмотрим для примера массив $[1, 4, 5 \parallel 2, 8, 9]$ (знак \parallel указывает точку раздела). 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.

Быстрая сортировка | Время выполнения в среднем случае: $O(n \log(n))$, в худшем случае: $O(n^2)$. Память: $O(\log(n))$.

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 через перестановки (см. далее).

Если мы будем много раз делить массив (и его подмассивы) относительно случайного элемента, то в результате получим отсортированный массив. Поскольку опорный элемент не 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) {           // Сортируем левую половину
4         quickSort(arr, left, index - 1);
5     }
6     if (index < right) {            // Сортируем правую половину
7         quickSort(arr, index, right);
8     }
9 }
10
11 int partition(int arr[], int left, int right) {
12     int pivot = arr[(left + right) / 2];    // Выбираем центральную точку
13     while (left <= right) {
14         // Находим элемент слева, который должен быть справа
15         while (arr[left] < pivot) left++;
16
17         // Находим элемент справа, который должен быть слева
18         while (arr[right] > pivot) right--;
19
20         // Меняем местами элементы и перемещаем левые и правые индексы
21         if (left <= right) {
22             swap(arr, left, right);        // Меняем местами элементы
23             left++;
24             right--;
25         }
26     }
27     return left;
28 }
```

Поразрядная сортировка | Время выполнения в среднем случае: $O(n \log(n))$, в худшем случае: $O(kn)$.

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 с использованием сравнения, которые в большинстве случаев не могут выполняться быстрее, чем за $O(\log(n))$, данный алгоритм

в худшем случае дает результат $O(kn)$, где n — количество элементов в массиве, а k — количество проходов алгоритма сортировки.

Алгоритмы поиска

Когда мы думаем об алгоритмах поиска, мы обычно подразумеваем бинарный поиск. Действительно, это очень полезный алгоритм. Заметьте, что хотя концепция довольно проста, разобраться во всех нюансах сложнее, чем кажется на первый взгляд. Изучите приведенный ниже код и обратите внимание на плюс и минус 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 }
```

Множество разновидностей поиска структур данных не ограничиваются вариациями бинарного поиска, поэтому приложите максимум усилий для их изучения. Можно, например, искать узел, используя бинарные деревья или хэш-таблицы. Не ограничивайте свои возможности!

Вопросы собеседования

- 10.1.** Дано: два отсортированных массива А и В. Размер массива А (свободное место в конце) позволяет поместить в него массив В. Напишите метод слияния массивов В и А, сохраняющий сортировку.

10.2. Напишите метод сортировки массива строк, при котором анаграммы группируются друг за другом.

10.3. Дано: отсортированный массив из n целых чисел, который был циклически сдвинут произвольное число раз. Напишите код для поиска элемента в массиве. Исходите из предположения, что массив изначально был отсортирован по возрастанию.

Пример:

Ввод: найдите индекс элемента со значением 5 в {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Вывод: 8.

10.4. Дано: файл размером 20 Гбайт, содержащий строки. Как выполнить сортировку такого файла?

10.5. Дано: отсортированный массив, состоящий из строк, разделенных пустыми строками. Напишите метод для обнаружения позиции заданной строки.

Пример:

Ввод: найдите индекс элемента "ball" в массиве {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Вывод: 4

10.6. Дано: матрица размером $M \times N$, строки и колонки которой отсортированы. Напишите метод нахождения элемента.

10.7. Цирк готовит новый аттракцион — пирамида из людей, стоящих на плечах друг у друга. Простая логика подсказывает, что люди, стоящие выше, должны быть ниже ростом и легче, чем люди, находящиеся в основании пирамиды. Учитывая информацию о росте и весе каждого человека, напишите метод, вычисляющий наибольшее 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)

10.8. Вы обрабатываете поток целых чисел. Периодически вам нужно находить ранг числа x (количество значений $\leq x$). 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 $\leq x$ (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`

Дополнительные вопросы: массивы и строки (1.3), рекурсия (8.3), модерирование (17.6, 17.12), вопросы повышенной сложности (18.5).

11. Масштабируемость и ограничения памяти

Несмотря на пугающее название, вопросы о масштабируемости чаще всего оказываются самыми легкими. Нет никаких ловушек и необычных алгоритмов (по крайней мере обычно их не бывает). Вам не нужно посещать спецкурсы по распределенным системам или иметь опыт системного проектирования. Любой умный программист с небольшим опытом может непринужденно ответить на все эти вопросы.

Пошаговый подход

Интервьюеры не пытаются проверить ваши знания в области системного проектирования. Фактически, интервьюеры чаще всего проверяют только знание основ информатики. Приведенная ниже последовательность действий отлично помогает при решении многих задач системного проектирования.

Шаг 1. 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 2. Вернитесь к реальности

Вернитесь к исходной задаче. Сколько данных можно разместить в одном компьютере? Какие проблемы могут возникнуть, если вы разделите данные? Обычно достаточно выяснить, как разделить данные и как дать понять одному из компьютеров, где находится недостающая информация.

Шаг 3. Решите задачу

Подумайте о том, как избавиться от обнаруженных проблем. Помните, что решение может полностью исключить проблему или немного улучшить ситуацию. Скорее всего, вам будет достаточно слегка модифицировать исходный алгоритм (шаг 1), но иногда его требуется существенно изменить.

Используйте итерационный подход. Как только проблемы, появившиеся на втором шаге, 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.

Что нужно знать: информация, стратегия и проблема

Типичная система

Хотя суперкомпьютеры еще не канули в Лету, многие веб-ориентированные компании используют огромные распределенные системы. Скорее всего, вам придется работать в такой системе.

Заполните следующую таблицу перед собеседованием, она поможет запомнить, сколько данных может находиться на компьютере.

Компонент	Типичная емкость/значение
Жесткий диск	
Оперативная память	
Пропускная способность интернет-канала	

Разделение данных

В некоторых случаях мы можем увеличить объем жесткого диска, рано или поздно мы дойдем до точки, когда данные придется распределять между несколькими компьютерами или жесткими дисками. Вопрос в следующем: какие именно данные и на какой компьютер или диск поместить? Есть несколько вариантов ответа:

В порядке появления

Мы можем просто разделить данные в хронологическом порядке. В этом случае, чтобы добавить еще один компьютер или диск, нам придется ждать момента, пока использующийся в данный момент компьютер или диск будет полностью заполнен. Преимущество этого способа — используется ровно столько компьютеров или дисков, сколько необходимо. Трудность в том, что в зависимости от задачи и набора данных таблица поиска может стать очень сложной и большой.

По значению хэш-функции

Альтернативный подход — хранение данных на машине, в соответствии со значением хэш-функции данных. Мы делаем следующее: 1) выбираем ключ для данных; 2) хэшируем ключ; 3) получаем остаток от деления этого значения на количество машин и 4) сохраняем данные на машине с номером, соответствующим получившемуся ранее значению. Таким образом, данные находятся на компьютере под номером $\#[\text{mod}(\text{hash}(\text{key}), N)]$.

Преимущество данного способа — таблица поиска не нужна. Каждая машина знает, где находится нужный фрагмент данных. Недостаток в том, что машина может получить слишком большой объем данных, превышающий ее возможности. В этом случае необходимо будет сделать балансировку (переместить часть данных на другой компьютер) или использовать древовидную структуру компьютеров (разделить данные на две машины).

По значению данных

Деление данных по значению хэш-функции никак не связано с типом данных и с тем, на какой машине они хранятся. В некоторых случаях мы можем сократить время отклика системы, используя информацию о представлении данных.

Допустим, вы разрабатываете социальную сеть. Друзья пользователей могут быть разбросаны по всему миру, но скорее всего, если пользователь живет в Мексике, то и большинство его друзей живут в Мексике, а не в России. Если «подобные» данные будут находиться на одной и той же машине, то поиск потребует меньше переходов между машинами.

Произвольно

Достаточно часто данные разбиваются произвольно, и мы используем таблицу поиска, чтобы идентифицировать, на какой машине находится та или иная часть информации. Этот способ требует большой таблицы поиска, но он упрощает некоторые аспекты проектирования системы и позволяет обеспечить лучшую балансировку.

Пример: найдите все документы, содержащие список слов

У вас есть миллион документов, и нужно найти все документы, содержащие определенный список слов. 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.

Шаг 1

На первом шаге нужно забыть о миллионах документов и притвориться, что вы должны обработать 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 создать индекс хэш-таблицы. Эта таблица выполняет преобразование слова в список документов, содержащих это слово.

```
"books" -> {doc2, doc3, doc6, doc8}  
"many" -> {doc1, doc3, doc7, doc8, doc9}
```

Чтобы найти документы, содержащие слова `many` и `books`, нам нужно заглянуть в таблицу 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 находиться на нескольких компьютерах. Более того, при определенных условиях — большое количество слов и их повторяемость в документах — полная хэш-таблица может не уместиться на одном компьютере. Что же делать в таких ситуациях?

1. Как мы будем делить нашу хэш-таблицу? Мы можем 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 хэш-таблица только для заданного подмножества документов.
2. Как только мы решим, каким образом разделить данные, необходимо обработать документ, находящийся на одной машине, и передать результаты на другую. Как работает этот процесс? (Если мы делим хэш-таблицу по документам, то этот шаг не нужен.)
3. Нам необходимо придумать метод, позволяющий узнать, какие данные и на каком компьютере будут храниться. Как выглядит таблица поиска и где она хранится? Это всего лишь три варианта, могут быть и другие.

Шаг 3

На третьем шаге мы занимаемся поиском решения каждой из перечисленных задач. Одно из возможных решений – упорядочить слова по алфавиту, чтобы каждый компьютер управлял определенным диапазоном слов (например, от `after` до `apple`).

Мы можем реализовать простой алгоритм проверки всех ключевых слов в алфавитном порядке, и при этом каждый компьютер будет загружен по максимуму. Когда ресурсы одного компьютера исчерпаны, мы переходим к следующему компьютеру.

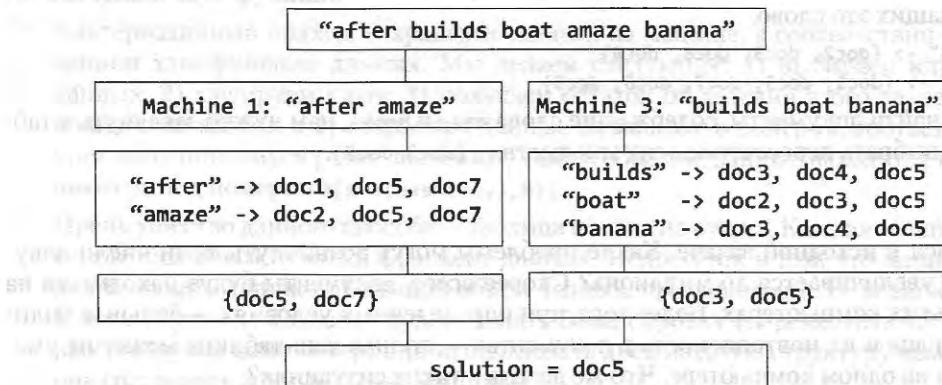
Преимущество этого подхода – простая и небольшая таблица поиска (так как в ней хранится только диапазон значений), при этом копии таблицы поиска могут сохраняться на каждой машине. Недостаток заключается в том, что при добавлении новых документов нам придется делать очень «дорогой» сдвиг ключевых слов.

Чтобы найти все документы, удовлетворяющие списку строк, нам нужно отсортировать список и затем передать каждому компьютеру поисковый запрос, учитывая информацию о словах, 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 сделает 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

- 11.1.** 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 службы и обоснуйте свой подход. Вы можете использовать любые технологии и любой механизм предоставления информации клиентским приложениям.

-
- дач.
ком-
вит-
огда
ру.
ней
кра-
вых
иро-
ин-
жно
").
пес-
слов
по-
- 11.2. Какие структуры данных вы бы стали использовать в очень больших социальных сетях вроде Facebook или LinkedIn? Опишите, как вы будете разрабатывать алгоритм, демонстрирующий круг знакомств или связь человека с человеком (Я → Боб → Сюзи → Джейсон → Ты).
- 11.3. Дан входной файл, содержащий четыре миллиарда целых чисел. Создайте алгоритм, генерирующий целое число, отсутствующее в файле. У вас есть 1 Гбайт памяти для этой задачи.
- Дополнительно:* а что если у вас есть всего 10 Мбайт?
- 11.4. Дано: массив целых чисел от 1 до n , где n не превышает 32 000. Значения в массиве могут повторяться, значение n — неизвестно. Выведите список всех повторяющихся элементов массива, имея в распоряжении 4 Кбайт оперативной памяти.
- 11.5. Как избежать зацикливаний при разработке поискового робота?
- 11.6. Дано: 10 миллиардов URL-адресов. Как обнаружить все дублирующиеся документы? Дубликатом считается совпадение URL-адреса.
- 11.7. Дано: веб-сервер самой простой поисковой системы. В системе есть 100 компьютеров, обрабатывающих поисковые запросы, которые могут генерировать запрос `processSearch(string query)` в другой кластер компьютеров, чтобы получить результат. Компьютер, отвечающий на запрос, выбирается случайным образом, вы не можете гарантировать, что одна и та же машина всегда будет получать один и тот же запрос. Метод `processSearch` очень затратный. Разработайте механизм кэширования новых запросов. Объясните, как при изменении данных будет обновляться кэш.

Дополнительные вопросы: объектно-ориентированное проектирование (7.7).

12. Тестирование

Прежде чем пройти мимо этой главы, пользуясь предлогом «я 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) тестирование программного кода; 4) устранение известной ошибки. Мы рассмотрим задачи всех четырех типов.

Помните, что все четыре типа задач предполагают, что вам придется учитывать ошибки ввода и «неидеальность» пользователя. Ждите подводных камней.

Чего ожидает интервьюер

Задания по тестированию предполагают, что вы напишете большой список тестов. До какой-то степени это верно, но вы должны придумать разумный список.

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 расставить приоритеты тестов должным образом? Вас могут попросить спроектировать электронную коммерческую систему, похожую на Amazon. Естественно, нужно убедиться, что изображения продуктов появляются на правильном месте, но еще более важно, чтобы корректно осуществлялись все платежи, продукты добавлялись в очередь на отгрузку и не было недовольных клиентов.
- **Знание, как правильно совместить все части:** понимаете ли вы, как работает программное обеспечение и как оно может вписаться в большую систему? Вас могут просить протестировать электронную таблицу Google. Очень важно не забыть проверить все этапы — открытие, сохранение и редактирование документов. Но электронная таблица Google — 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 говорят 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 «глючит», когда он открывает какое-либо изображение, а вы ему советуете переустановить программу — это не лучший совет. Тестирование должно быть выполнимым и реалистичным для компании.

Демонстрация владения перечисленными навыками покажет, что вы можете стать ценным членом команды тестеров.

Тестирование реального объекта

Некоторые кандидаты очень удивляются, когда их просят протестировать ручку. Ведь они хотят стать тестерами программного обеспечения. 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 будет трансформировать ее в скульптурную форму животного». И в том и в другом случае используется скрепка, так что ответ на вопрос даст общее представление о требующихся тестах.

Шаг 2: каковы варианты использования?

Для вас будет полезно составить список возможных вариантов использования. В рассматриваемом случае скрепка используется для скрепления листов без повреждения бумаги.

Возможно, вам придется тестировать отправку и получение контента, запись и стирание и т. д.

Шаг 3: каковы граници действия предмета?

Ограничением в нашем случае может быть, например, до 30 листов бумаги без повреждений скрепки и до 50 листов с небольшими повреждениями.

Границы распространяются и на окружающую среду. Должна ли скрепка работать при высоких температурах (90–110 градусов)? А как насчет холода?

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 отказ-неисправность — одежда окажется плохо постирана. При загрузке более 45 рубашек может быть отказ — потеря работоспособности, но в этом случае отказ означает, что машина просто не запустится. Отказ не предполагает взрыва или пожара.

Шаг 5: как осуществить тестирование?

В некоторых случаях необходимо обсудить нюансы процедуры тестирования. Если вам нужно убедиться, что стул может прослужить 5 лет, не будете же вы брать его домой и ждать этот срок? 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. Существенное различие заключается в том, что при тестировании программного обеспечения больший акцент ставится на деталях самой процедуры тестирования.

Перед началом тестирования программного обеспечения нужно обратить внимание на два момента.

- **Ручное vs автоматическое 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 все задачи. Не забывайте, что компьютер может распознать только то, что было запрограммировано заранее, а человек может заметить и новые проблемы, которые не были учтены. И люди, и компьютеры являются неотъемлемыми частями процесса тестирования.
- **Ящики: «черный» vs «белый».** Различие «черный»/«белый» указывает 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 «черного» ящика также можно автоматизировать, но это сделать существенно сложнее.

Давайте рассмотрим последовательность наших действий от начала до конца.

Шаг 1: какой — «черный» или «белый» — ящик использовать?

Этот вопрос можно задать и позднее, но мне нравится задавать его в самом начале. Согласуйте с интервьюером метод тестирования — «черный» или «белый» ящик (или оба).

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 стороны — их действия не должны быть ограничены блокировкой.

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. 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 блокиро-

вать 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 срабатываний является 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: каковы тестовые прецеденты? Как осуществить тестирование?

Вот тут 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 тестиировать? Какой из этих шагов можно автоматизировать, а какой требует человеческого вмешательства?

Помните, что автоматизация позволяет провести очень серьезное тестирование, но все есть существенные недостатки. Ручное тестирование обязательно должно входить в процедуру тестирования, которую вы предлагаете.

Пройдитесь по списку и задумайтесь. Используйте системный подход. Разбейте программный продукт на основные компоненты и тестируйте их по-отдельности. Работая в этом направлении, вы не только составите расширенный список прецедентов, но и покажете, что вы способны действовать системно.

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. Вы должны обсудить все предположения, особенно затрагивающие работу в определенных ситуациях.

Предположим, что вас попросили написать код для проверки функции `sort(int[] array)`, выполняющей сортировку массива целых чисел. Действуйте по плану.

Шаг 1: определите тестовые случаи

Проанализируйте следующие типы тестовых случаев:

- Нормальный случай. Функция генерирует правильный результат для типичного ввода? Не забудьте о потенциальных проблемах. Например, если сортировка часто требует разделений, разумно предположить, что алгоритм может перестать работать на массивах с нечетным количеством элементов, так как их невозможно разделить ровно пополам.
- Аномальные значения. Что происходит, если ваш массив окажется пустым? Или очень маленьким (1 элемент)? 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, что произойдет, если значение n окажется отрицательным?
- Странный ввод. Что случится, если функция получит отсортированный массив? Или отсортированный в обратном порядке массив?

Проведение таких тестов требует знания функции, которую вы пишете. Если вам не ясны ограничения, поговорите о них с интервьюером.

Шаг 2: определите ожидаемый результат

Ожидаемый результат очевиден — правильный вывод. Однако в некоторых случаях нужно проверить множество дополнительных деталей. Например, если метод `sort` возвращает новый отсортированный массив, вы должны проверить, не изменился ли исходный массив.

Шаг 3: напишите тестовый код

Как только вы определились с тестовыми случаями и результатам, нужно написать максимально простой код, реализующий тестовые случаи:

```
1 void testAddThreeSorted() {  
2     myList list = new myList();  
3     list.addThreeSorted(3, 1, 2); // Добавляем три элемента  
                                // в отсортированном порядке  
4     assertEquals(list.getElement(0), 1);  
5     assertEquals(list.getElement(1), 2);  
6     assertEquals(list.getElement(2), 3);  
7 }
```

Поиск и устранение неисправностей

Как бы вы произвели процесс устранения существующей проблемы? Много кандидатов, получив подобный вопрос, дают неправильный ответ вроде «переустановлю программу». Эту задачу также можно решить, используя структурный подход.

Давайте рассмотрим пример. Вы — член команды Google Chrome — получили отчет об ошибке: браузер отказывается запускаться. Ваша действия?

Переустановка браузера может решить проблему этого пользователя, но вряд ли поможет другим, у которых могла возникнуть аналогичная ситуация. Ваша цель — разобраться, что происходит, так, чтобы разработчики могли потом исправить ошибку.

Шаг 1: разберите сценарий

Прежде всего нужно задать вопросы, позволяющие максимально прояснить ситуацию:

- Когда появилась данная проблема?
- Какая версия браузера используется? Какая операционная система?
- Проблема возникает постоянно? Как часто она возникает? Когда именно она происходит?
- Есть ли сообщение об ошибке?

Шаг 2: разбейте задачу на части

Теперь, когда вам ясны детали сценария, вы можете попробовать разбить задачу на части. В нашем случае:

1. Переидите в стартовое меню Windows.
2. Щелкните на значке Chrome.
3. Запустите экземпляр браузера.
4. Браузер загружает настройки.
5. Браузер отправляет HTTP-запрос на получение домашней страницы.
6. Браузер получает HTTP-ответ.
7. Браузер анализирует 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, в чем причина.

Шаг 3: создайте конкретные, управляемые тесты

Для проверки каждого компонента должны быть определенные инструкции. Вы можете попросить пользователя выполнить их. В реальном мире вы будете иметь дело с клиентами и не должны давать им инструкции, которые они не смогут выполнить.

Вопросы собеседования

- 12.1. Найдите ошибку (ошибки) в следующем коде:

```
1 unsigned int i;  
2 for (i = 100; i >= 0; --i)  
3 printf("%d\n", i);
```

- 12.2. У вас есть 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 падение приложения? Как вы проверите каждую?

- 12.3. Дано: игра «Шахматы». В ней реализован 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.

- 12.4. Как вы проведете нагружочный тест веб-страницы без использования специальных инструментов тестирования?

- 12.5. Как вы будете тестировать авторучку?

- 12.6. Как вы будете тестировать банкомат, подключенный к распределенной банковской системе?

13. Си С++

Хороший интервьюер не станет требовать, чтобы вы написали код на языке, которого вы не знаете. Если вас просят написать код на C++, значит, вы упомянули его в резюме. Если вы не помните все API, не волнуйтесь, большинство интервьюеров (но не все) не потребуют этого. Я рекомендую изучить базовый синтаксис C++ так, чтобы вы могли непринужденно ответить на вопросы по C/C++.

Классы и наследование

Классы C++ подобны классам в других языках программирования, но мы все же рассмотрим их синтаксис. Следующий код демонстрирует реализацию базового класса с наследованием.

```

1 #include <iostream>
2 using namespace std;
3
4 #define NAME_SIZE 50      // Определяем макрос
5
6 class Person {
7     int id;                // по умолчанию все члены приватные
8     char name[NAME_SIZE];
9
10    public:
11        void aboutMe() {
12            cout << "Я - личность";
13        }
14    };
15
16 class Student : public Person {
17     void aboutMe() {
18         cout << "Я - студент";
19     }
20 };
21
22 int main() {
23     Student * p = new Student();
24     p->aboutMe();          // выводит текст "Я - студент"
25     delete p;              // Внимание! Убедитесь, что освободили выделенную память
26
27 }
```

По умолчанию в C++ все члены и методы приватные. Изменить это можно с помощью ключевого слова `public`.

Конструкторы и деструкторы

Конструктор – это класс, который автоматически вызывается при создании объекта. Если конструктор не определен, компилятор автоматически генерирует так называемый конструктор по умолчанию.

```
1 Person(int a) {
2     id = a;
3 }
```

Поля класса необходимо инициализировать:

```
1 Person(int a) : id(a) {
2     ...
3 }
```

Член класса `id` инициализируется перед созданием объекта и перед выполнением оставшейся части конструктора. Это полезно в том случае, когда нужно создать постоянные поля-константы, присвоить значение которым нужно только один раз.

Деструктор удаляет объект и автоматически вызывается при уничтожении объекта. Мы не можем передать аргумент деструктору, поскольку мы не вызываем его явно:

```
1 ~Person() {
2     delete name; // очищаем память, выделенную для name
3 }
```

Виртуальные функции

Определим новый объект `p` класса `Student`:

```
1 Student * p = new Student();
2 p->aboutMe();
```

Что случится, если мы определили `p` как класс `Person *`?

```
1 Person * p = new Student();
2 p->aboutMe();
```

В этом случае будет выведен текст «Я – личность». Это связано с тем, что имя функции `aboutMe` разрешается использовать во время компиляции. Такой механизм называется статической линковкой. Если вы хотите убедиться, что будет вызвана реализация `aboutMe` из класса `Student`, функцию нужно сделать виртуальной (`virtual`).

```
1 class Person {
2     ...
3     virtual void aboutMe() {
4         cout << "Я - личность";
5     }
6 };
7
8 class Student : public Person {
9     public:
10    void aboutMe() {
11        cout << "Я - студент";
12    }
13};
```

Существует и другое применение виртуальных функций, например когда мы не можем или не хотим реализовать метод для родительского класса. Представьте, например, что нам нужно наследовать классы `Student` и `Teacher` от класса `Person` так, чтобы определить общий метод `addCourse (string s)`. Однако вызов `addCourse` для класса `Person` не имеет смысла, поскольку этот метод должен быть привязан к классам `Student` или `Teacher`.

В этом случае в классе `addCourse` нужно определить `addCourse` как виртуальную функцию, но это всего лишь заглушка, а реальный код будет описан в подклассах:

```

1 class Person {
2     int id; // по умолчанию все члены приватные
3     char name[NAME_SIZE];
4     public:
5         virtual void aboutMe() {
6             cout << "Я - личность" << endl;
7         }
8         virtual bool addCourse(string s) = 0;
9     };
10
11 class Student : public Person {
12     public:
13         void aboutMe() {
14             cout << "Я - студент" << endl;
15         }
16
17         bool addCourse(string s) {
18             cout << "Добавлен курс " << s << " в объект Student" << endl;
19             return true;
20         }
21     };
22
23 int main() {
24     Person * p = new Student();
25     p->aboutMe(); // выведет "Я - студент"
26     p->addCourse("История");
27     delete p;
28 }
```

Заметьте, что определяя `addCourse` как виртуальную функцию, мы делаем класс `Person` настоящим абстрактным классом и не можем инстанцировать его.

Виртуальный деструктор

Виртуальная функция была своеобразным предисловием к «виртуальному деструктору». Вот как можно реализовать деструкторы для `Person` и `Student`:

```

1 class Person {
2     public:
3     ~Person() {
4         cout << "Удаляем Person" << endl;
5     }
6 };
7
```

```

8 class Student : public Person {
9     public:
10     ~Student() {
11         cout << "Удаляем Student." << endl;
12     }
13 };
14
15 int main() {
16     Person * p = new Student();
17     delete p; // выведет "Удаляем Person."
18 }

```

Школьку `p` — это объект класса `Person` (как было показано ранее), будет вызван деструктор этого класса. Но это затруднительно, поскольку память для объекта класса `Student` еще не освобождена. Чтобы исправить это, нужно просто объявить деструктор для `Person` виртуальным:

```

1 class Person {
2     public:
3     virtual ~Person() {
4         cout << "Удаляем Person." << endl;
5     }
6 };
7
8 class Student : public Person {
9     public:
10    ~Student() {
11        cout << "Удаляем Student." << endl;
12    }
13 };
14
15 int main() {
16     Person * p = new Student();
17     delete p;
18 }

```

Выход будет следующим:

```

Удаляем Student.
Удаляем Person.

```

Значения по умолчанию

Вы можете задавать значения для аргументов функции по умолчанию. Обратите внимание, что если вам нужны обычные параметры и параметры со значениями по умолчанию, то последние должны располагаться правее, иначе становится не очевидно, что некоторые параметры можно не указывать:

```

1 int func(int a, int b = 3) {
2     x = a;
3     y = b;
4     return a + b;
5 }
6
7 w = func(4);
8 z = func(4, 5);

```

Перезагрузка операторов

Перезагрузка операторов позволяет применять любые операторы (например, `+`) к объектам, которые изначально не поддерживали данные действия. Если вам нужно объединить два объекта `BookShelves` в один, вы можете использовать оператор `+`:

```
1 BookShelf BookShelf::operator+(Packet &other) { ... }
```

Указатели и ссылки

Указатель хранит адрес переменной и может быть использован для осуществления операций над переменной (например, чтение и изменение ее значения). Если два указателя идентичны, то изменение значения по одному из них приведет к изменению другого (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 второе имя (псевдоним) существующего объекта, для ссылки не выделяется память. Например:

```
1 int a = 5;
2 int & b = a;
3 b = 7;
4 cout << a; // выведет 7
```

Во второй строке `b` — это ссылка на `a`. Изменяя `b`, мы изменяем также `a`.

Вы не можете создать ссылку без указания конкретного места, на которое она будет ссылаться. Однако вы можете создать ссылку, указывающую на определенное значение (без переменной):

```
1 /* выделяем место в памяти для значения 12 и
2 делаем ссылку b на этот адрес памяти */
3 int & b = 12;
```

В отличие от указателей, ссылки не могут принимать значение `NULL` и не могут переназначаться на другой участок памяти.

Арифметика указателей

Часто программисты используют 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 4 байта, следовательно, указатель переходит на следующую позицию — следующий элемент массива. При работе со структурами данных других типов переход осуществляется на столько байтов, сколько занимает структура данных.

Шаблоны

Шаблоны — это возможность применить класс к данным другого типа. Представьте, что существует структура данных, напоминающая список, которую нужно использовать для данных разных типов. Следующий код реализует класс `ShiftedList`:

```
1 template <class T>
2 class ShiftedList {
3     T* array;
4     int offset, size;
5 public:
6     ShiftedList(int sz) : offset(0), size(sz) {
7         array = new T[size];
8     }
9
10    ~ShiftedList() {
11        delete [] array;
12    }
13
14    void shiftBy(int n) {
15        offset = (offset + n) % size;
16    }
17
18    T getAt(int i) {
19        return array[convertIndex(i)];
20    }
21
22    void setAt(T item, int i) {
23        array[convertIndex(i)] = item;
24    }
25
26    private:
27    int convertIndex(int i) {
28        int index = (i - offset) % size;
29        while (index < 0) index += size;
30        return index;
31    }
32 }
33
34 int main() {
35     int size = 4;
36     ShiftedList<int> * list = new ShiftedList<int>(size);
37     for (int i = 0; i < size; i++) {
38         list->setAt(i, i);
39     }
40     cout << list->getAt(0) << endl;
```

продолжение ↗

```

41     cout << list->getAt(1) << endl;
42     list->shiftBy(1);
43     cout << list->getAt(0) << endl;
44     cout << list->getAt(1) << endl;
45     delete list;
46 }

```

Вопросы собеседования

- 13.1.** Напишите на C++ метод, выводящий последние k строк входного файла.
- 13.2.** Сопоставьте хэш-таблицу и `map` из стандартной библиотеки шаблонов (STL). Как организована хэш-таблица? Какая структура данных будет оптимальной для небольших объемов данных?
- 13.3.** Как работают виртуальные функции в C++?
- 13.4.** Какая разница между глубоким и поверхностным копированием? Объясните, как использовать эти виды копирования.
- 13.5.** Каково назначение ключевого слова `volatile` в C?
- 13.6.** Почему деструктор базового класса должен объявляться виртуальным?
- 13.7.** Напишите метод, получающий указатель на структуру `Node` как параметр и возвращающий полную копию переданной структуры данных. Структура данных `Node` содержит два указателя на другие узлы (другие структуры `Nodes`).
- 13.8.** Напишите класс для интеллектуального указателя. Это тип данных (обычно использующий шаблоны), симулирующий указатель и производящий автоматический сбор мусора. Он автоматически подсчитывает количество ссылок на объект `SmartPointer<T*>` и освобождает объект типа T, когда число ссылок становится равным 0.
- 13.9.** Напишите функцию динамического распределения памяти, работающую с адресами, кратными степеням двойки.
Пример:
`align_malloc(1000, 128)` возвращает адрес памяти, который является кратным 128 и указывает на память размером 1000 байт.
Функция `aligned_free()` освобождает память, выделенную с помощью `align_malloc`.
- 13.10.** Напишите на C функцию (`my2DAlloc`), которая выделяет память для двумерного массива. Минимизируйте количество вызовов функции и убедитесь, что к памяти можно обращаться как `arr[i][j]`.

Дополнительные вопросы: массивы и строки (1.2), связные списки (2.7), тестирование (12.1), Java (14.4), потоки и блокировки (16.3).

14. Java

Данная глава целиком посвящена синтаксису Java. Вопросы по синтаксису языка программирования редко встречаются на собеседованиях в крупных компаниях, которые любят тестировать 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 не существует специального подхода к подготовке. Согласитесь, ничто не может заменить правильный ответ. Конечно, лучше всего просто выучить Java. Но если вам нужно подготовиться к собеседованию, используйте следующий подход:

- 1 Создайте примерный сценарий и разберитесь, что существенно, а что — нет.
- 2 Подумайте, как реализовать сценарий на других языках.
- 3 Взгляните на ситуацию как разработчик языка. Проанализируйте причинно-следственные связи.

Ваш ответ может произвести впечатление на интервьюера, а может и не понравиться ему, если вы отвечаете заученно. 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...»

Ключевое слово final

Ключевое слово `final` в Java меняет значение в зависимости от того, к чему оно применяется: к переменной, классу или методу.

- Переменная: значение переменной не может быть изменено после инициализации.
- Метод: метод не может быть переопределен подклассом.
- Класс: класс не может иметь подкласс.

Ключевое слово finally

Ключевое слово `finally` используется вместе с `try/catch` и гарантирует, что секция `finally` будет выполнена, даже если произойдет исключительная ситуация. Блок `finally` выполняется после блоков `try/catch`, но перед возвратом управления.

Посмотрим, как `finally` работает на практике.

```
1 public static String lem() {  
2     System.out.println("lem");  
3     return "возврат из lem";  
4 }  
5  
6 public static String foo() {
```

продолжение ↴

```

7     int x = 0;
8     int y = 5;
9     try {
10         System.out.println("запуск try");
11         int b = y / x;
12         System.out.println("завершение try");
13         return "возврат из try";
14     } catch (Exception ex) {
15         System.out.println("catch");
16         return lem() + " | возврат из catch";
17     } finally {
18         System.out.println("finally");
19     }
20 }
21
22 public static void bar() {
23     System.out.println("запуск bar");
24     String v = foo();
25     System.out.println(v);
26     System.out.println("завершение bar");
27 }
28
29 public static void main(String[] args) {
30     bar();
31 }

```

Выведено будет:

```

1 запуск bar
2 запуск try
3 catch
4 lem
5 finally
6 возврат из lem | возврат из catch
7 завершение bar

```

Посмотрите внимательно на строки вывода 3 и 5. Блок catch полностью выполнен (включая вызов функции в операторе return), затем блок finally, а затем — возврат из функции.

Метод finalize

Автоматический уборщик мусора вызывает метод finalize() перед уничтожением объекта. Класс может перезаписать метод finalize(), чтобы определить собственную процедуру уборки мусора.

```

1 protected void finalize() throws Throwable {
2     /* Закрываем файлы, освобождаем ресурсы и т. д. */
3 }

```

Перегрузка vs переопределение

Перегрузка — 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 возникает, когда метод разделяет имя с функцией или другим методом в суперклассе.

```
1 public abstract class Shape {
2     public void printMe() {
3         System.out.println("Я - фигура");
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("Я - круг.");
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 Я - круг.
- 2 78.75
- 3 Я - фигура.
- 4 10.0

Заметьте, что Circle переопределил printMe(), тогда как Ambiguous оставил этот метод как есть.

Java Collection Framework

Java Collection Framework — очень полезный набор связанных классов и интерфейсов, в чем вы еще убедитесь, читая эту книгу. Давайте рассмотрим его наиболее полезные элементы.

ArrayList — массив с динамически изменяемым размером, он автоматически расширяется 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.

```
1 Vector<String> myVect = new Vector<String>();
2 myVect.add("one");
3 myVect.add("two");
4 System.out.println(myVect.get(0));
```

LinkedList позволяет реализовать связный список на Java, о нем редко вспоминают на собеседованиях, но он используется для демонстрации синтаксиса итераторов.

```
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 работе.

```
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 разобрались с синтаксисом языка.

Вопросы собеседования

Обратите внимание, что практически все решения в этой книге написаны на Java, поэтому вопросов для этой главы немного. Кроме того, большинство этих вопросов затрагивает «нюансы» синтаксиса, тогда как остальная часть книги содержит вопросы о программировании на Java.

- 14.1.** Как повлияет на наследование объявление конструктора приватным?
- 14.2.** Будет ли выполняться блок `finally` (на Java), если оператор `return` находится внутри `try`-блока (`try-catch-finally`)?
- 14.3.** В чем разница между `final`, `finally` и `finalize`?
- 14.4.** Объясните разницу между шаблонами в C++ и обобщениями (`generic`) в Java.
- 14.5.** Объясните, что такое рефлексия объекта в Java и когда она используется.
- 14.6.** Реализуйте класс `CircularArray` для массива, хранящего структуру данных, обеспечивающий эффективный циклический сдвиг. Класс должен использовать обобщенный тип и поддерживать итерацию через стандартный оператор `for(Obj o : circularArray)`.

Дополнительные вопросы: массивы и строки (#1.4), объектно-ориентированное проектирование (#8.10), потоки и блокировки (#16.3).

```
SELECT Students.StudentName, Students
  FROM Students WHERE 1020 <= StudentID
  JOIN Students ON Students.StudentID = Students.ID
  JOIN Courses ON Courses.CourseID = Students.StudentID
  GROUP BY Students.StudentID;
```

этого варианта имеются три ошибки.

Мы исключили студентов, которые не получали оценки по предметам, поскольку «бесценные» студенты не могут быть членами группы. Нужно изменить «`where`» на «`where Courses.CourseID = Students.StudentID`». Правильный запрос выглядит так:

```
SELECT Students.StudentName, Students
  FROM Students WHERE 1020 <= StudentID
  JOIN Students ON Students.StudentID = Students.ID
  JOIN Courses ON Courses.CourseID = Students.StudentID
  GROUP BY Students.StudentID;
```

Задача №6
Ход решения:
Мы должны определить, каким образом можно определить, что студент не является членом группы. Для этого мы можем использовать функцию `count` от той же самой таблицы, которая содержит информацию о студентах, и это значение должно быть равно количеству предметов, на которых учащийся имеет оценку «отлично» (то есть 5). Итак, мы можем написать следующий запрос:

15. Базы данных

Кандидатов, которые написали в резюме, что имеют опыт работы с базами данных, могут попросить продемонстрировать свои знания на примере реализации SQL-запросов или разработки базы данных для приложения. Давайте рассмотрим некоторые ключевые понятия и проведем краткий обзор решения задач такого типа.

Когда вы смотрите на приведенные здесь запросы, не удивляйтесь незначительным различиям в синтаксисе. Существует множество вариантов SQL, и вы, возможно, работали с другой версией языка. Все примеры, приведенные в этой книге, были проверены в Microsoft SQL Server.

SQL-синтаксис и его варианты

Разработчики, как правило, используют в SQL-запросах явное (explicit) и неявное (implicit) объединения:

```
1 /* Явное объединение */
2 SELECT CourseName, TeacherName
3   FROM Courses INNER JOIN Teachers
4     ON Courses.TeacherID = Teachers.TeacherID
5
6 /* Неявное объединение */
7 SELECT CourseName, TeacherName
8   FROM Courses, Teachers
9     WHERE Courses.TeacherID = Teachers.TeacherID
```

Оба варианта эквивалентны, и не имеет значения, какой вы выберете. В этой книге будет использоваться явное объединение.

Денормализованные и нормализованные базы данных

Нормализованные базы данных позволяют минимизировать избыточность, а денормализованные базы данных проектируются, чтобы оптимизировать время 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 — более высокая «стоимость»¹ объединений.

Денормализованные базы данных хранят избыточные записи. Если мы знаем, что один запрос часто повторяется², то целесообразно будет хранить имя преподавателя в таблице `Courses`. Денормализованные базы данных используются для создания хорошо масштабируемых систем.

¹ Время (чтения из базы данных) — «деньги». — Примеч. перев.

² Например, выборка имени преподавателя по его `TeacherID`. — Примеч. перев.

SQL-операторы

Давайте рассмотрим основной синтаксис SQL на примере упомянутой ранее базы данных. Структура этой базы данных очень проста (* указывает на первичный ключ):

Courses: CourseID*, CourseName, TeacherID

Teachers: TeacherID*, TeacherName

Students: StudentID*, StudentName

StudentCourses: CourseID*, StudentID*

Используя вышеприведенные таблицы, мы реализуем следующие запросы.

Запрос 1: регистрация студента

Реализуйте запрос, позволяющий получить список всех студентов и количество курсов, которые посещает каждый студент.

Сначала мы могли бы реализовать это так:

```
1 /* Некорректный 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 варианта имеются три проблемы:

1. Мы исключили студентов, которые не зарегистрировались ни на каких курсах, поскольку объединение `StudentCourses` содержит только зарегистрированных студентов. Нужно изменить объединение на `LEFT JOIN`.
2. Даже если изменить объединение 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 остаются повторяющиеся `StudentName` в каждой группе. Как база данных узнает, какое `StudentName` вернуть? Несомненно, значение может оставаться тем же, но база данных не понимает этого. Мы должны применить функцию-агрегатор, например `first(Students.StudentName)`.

Таким образом, запрос примет вид:

```
1 /* Решение 1: вспомогательный запрос */
2 SELECT StudentName, Students.StudentID, Cnt
3   FROM (
4     SELECT Students.StudentID,
5           count(StudentCourses.CourseID) as [Cnt]
6     FROM Students LEFT JOIN StudentCourses
7       ON Students.StudentID = StudentCourses.StudentID
8     GROUP BY Students.StudentID
9   ) T INNER JOIN Students on T.studentID = Students.StudentID
```

Посмотрев на этот код, вы можете спросить, почему бы не выбрать имя студента в третьей строке и избавиться от строк 3–6. Это (ошибочное) решение приведено ниже.

```
1 /* Неправильный код */
2 SELECT StudentName, Students.StudentID,
3     count(StudentCourses.CourseID) as [Cnt]
4     FROM Students LEFT JOIN StudentCourses
5     ON Students.StudentID = bookStudentCourses.StudentID
6     GROUP BY Students.StudentID
```

Так поступать нельзя. Можно выбирать только значения, находящиеся в функции-агрегаторе или же в операторе GROUP BY.

Эту же задачу можно решить по-другому:

```
1 /* Решение 2: Добавляем StudentName оператор GROUP BY */
2 SELECT StudentName, Students.StudentID,
3     count(StudentCourses.CourseID) as [Cnt]
4     FROM Students LEFT JOIN StudentCourses
5     ON Students.StudentID = StudentCourses.StudentID
6     GROUP BY Students.StudentID, Students.StudentName
```

или

```
1 /* Решение 3: функция-агрегатор. */
2 SELECT max(StudentName) as [StudentName], bookStudents.StudentID,
3     count(bookStudentCourses.CourseID) as [Count]
4     FROM bookStudents LEFT JOIN bookStudentCourses
5     ON bookStudents.StudentID = bookStudentCourses.StudentID
6     GROUP BY bookStudents.StudentID
```

Запрос 2: размер аудитории

Реализуйте запрос, позволяющий получить список всех преподавателей и количество студентов, которым преподает каждый из них. Отсортируйте список в порядке убывания количества студентов.

Давайте попробуем построить данный запрос пошагово. Прежде всего, нужно получить список TeacherID и узнать, сколько студентов связано с каждым TeacherID. Это очень напоминает предыдущий случай.

```
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, что INNER JOIN не выбирает преподавателей без аудитории. 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, как мы обработали NULL-значения в SELECT: NULL-значение конвертируется в 0.

Проектирование небольшой базы данных

На собеседовании вас могут попросить разработать собственную базу данных. Давайте рассмотрим этот процесс поподробнее. Обратите внимание, что между этой задачей и объектно-ориентированным проектированием есть много общего.

Шаг 1: уберите неоднозначности

В вопросах по базам данных часто присутствуют неоднозначности (умышленно или неумышленно). Прежде чем начать проектировать базу данных, вам нужно разобраться с этими проблемами.

Допустим, вас попросили разработать систему хранения информации для агентства 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 условия (например, дублирование контактной информации человека в БД) должны учитываться в процессе работы.

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, какими должны быть таблицы и как эти таблицы соотносятся между собой. Это будут отношения «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:

Buildings		
BuildingID	BuildingName	BuildingAddress

Apartments		
ApartmentID	ApartmentAddress	BuildingID

Обратите внимание, что таблица квартир (**Apartments**) связана с таблицами зданий через поле **BuildingID**.

Если мы хотим учсть ситуацию, когда один и тот же человек арендует больше чем одну квартиру, нужно использовать отношение «многие-ко-многим» следующим образом:

Tenants		
TenantID	TenantName	TenantAddress
Apartments		
ApartmentID	ApartmentAddress	BuildingID
TenantApartment		
TenantID	ApartmentID	

В таблице TenantApartments хранятся отношения между Tenants и Apartments.

Шаг 4: исследуйте действия

Осталось только разобраться с деталями. «Пройдитесь» по основным операциям, которые планируется осуществлять, чтобы разобраться, как сохранять и получать данные. Не забудьте, что придется обрабатывать информацию об условиях аренды, ценах, выездах на осмотр и т. д. Каждое из этих действий потребует дополнительных таблиц и полей.

Проектирование больших баз данных

Когда проектируется большая, масштабная база данных, использованные в предыдущих примерах объединения (join) работают очень медленно. Вам придется денормализовать данные. Подумайте о том, как будут использоваться данные. Возможно, их придется продублировать в нескольких таблицах.

Вопросы собеседования

Вопросы 1–3 относятся к следующей базе данных:

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar
BuildingID	int	BuildingName	varchar		
		Address	varchar		

Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

Обратите внимание, что 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).

- 15.1. Напишите SQL-запрос для получения списка арендаторов, которые снимают более одной квартиры.
- 15.2. Напишите SQL-запрос для получения списка всех зданий и количества открытых запросов (запросов со статусом *open*).
- 15.3. Дом № 11 находится на капитальном ремонте. Напишите запрос, который закрывает все запросы на квартиры в этом здании.
- 15.4. Какие существуют типы связей? Объясните, чем они различаются и почему определенные типы лучше подходят для конкретных ситуаций.
- 15.5. Что такое денормализация? Поясните этот процесс.
- 15.6. Нарисуйте диаграмму связей для базы данных, в которой фигурируют компании, клиенты и сотрудники компаний.
- 15.7. Разработайте простую базу данных, содержащую данные об успеваемости студентов, и напишите запрос, возвращающий список лучших студентов (лучшие 10 %), отсортированный по их среднему баллу.

Дополнительные вопросы: объектно-ориентированное проектирование (8.6).

другой вариант – расширение класса *Thread*. Это предпочтительнее, чем использовать метод *run()*. Тогда подкласс унаследует все методы класса *Thread*, включая конструктор.

Следующий фрагмент кода демонстрирует создание нового класса, наследующего класс *Thread* и переопределяющего метод *run()*:

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Hello from my thread!");
    }
}
```

Если вы хотите, чтобы ваша программа не останавливалась, то вам нужно будет запустить новый поток выполнения:

```
MyThread mt = new MyThread();
mt.start();
```

или же перехватить исключение:

```
try {
    mt.start();
} catch (InterruptedException e) {
    System.out.println("Thread interrupted");
}
```

```
public class Example {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                System.out.println("Hello from thread!");
            }
        });
        t.start();
    }
}
```

16. Потоки и блокировки

На собеседовании в Microsoft, Google или Amazon много внимания навыкам реализации поточных алгоритмов не уделяют (если только вы не будете работать в команде, для которой важен этот вопрос). Однако интервьюеры всегда стараются оценить, насколько вы разбираетесь в потоках, 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 мы обсудим оба способа.

Реализация интерфейса `Runnable`

Интерфейс `Runnable` имеет следующую структуру:

```
1 public interface Runnable {  
2     void run();  
3 }
```

Для создания и использования потока с использованием этого интерфейса нужно:

1. Создать класс, который реализует интерфейс `Runnable`. Объект этого класса — это `Runnable`-объект.
2. Создать объект типа `Thread`, передав `Runnable`-объект в качестве аргумента конструктору `Thread`. Объект `Thread` теперь обладает `Runnable`-объектом, который реализует метод `run()`.
3. Метод `start()` вызывается объектом `Thread`, созданным ранее.

Пример:

```
1 public class RunnableThreadExample implements Runnable {  
2     public int count = 0;  
3  
4     public void run() {  
5         System.out.println("Запускается RunnableThread.");  
6         try {  
7             while (count < 5) {  
8                 Thread.sleep(500);  
9                 count++;  
10            }  
11        } catch (InterruptedException exc) {  
12            System.out.println("RunnableThread прерван.");  
13        }  
14        System.out.println("RunnableThread завершен.");  
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     /* ждет, пока поток медленно посчитает до 5 */
24     while (instance.count != 5) {
25         try {
26             Thread.sleep(250);
27         } catch (InterruptedException exc) {
28             exc.printStackTrace();
29         }
30     }
31 }

```

Обратите внимание, что в данном 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 вариант – расширение класса `Thread`. Это почти всегда означает, что мы переопределяем метод `run()`. Тогда подкласс сможет явно вызывать конструктор потока в его конструкторе.

Следующий код приводит пример расширения класса `Thread`:

```

1 public class ThreadExample extends Thread {
2     int count = 0;
3
4     public void run() {
5         System.out.println("Поток запущен.");
6         try {
7             while (count < 5) {
8                 Thread.sleep(500);
9                 System.out.println("В потоке, счетчик = " + count);
10                count++;
11            }
12        } catch (InterruptedException exc) {
13            System.out.println("Поток прерван.");
14        }
15        System.out.println("Поток завершен.");
16    }
17 }
18
19 public class ExampleB {
20     public static void main(String args[]) {
21         ThreadExample instance = new ThreadExample();
22         instance.start();
23     }
}

```

продолжение

```

24     while (instance.count != 5) {
25         try {
26             Thread.sleep(250);
27         } catch (InterruptedException exc) {
28             exc.printStackTrace();
29         }
30     }
31 }
32 }
```

Этот код очень похож на предыдущий вариант. Отличие в том, что мы расширяем класс `Thread` вместо того, чтобы реализовывать интерфейс. В результате мы можем вызвать `start()` в экземпляре класса.

Расширение класса `Thread` vs реализация Runnable-интерфейса

Существуют две причины, по которым реализация `Runnable`-интерфейса предпочтительнее расширения класса `Thread`:

1. Java не поддерживает множественное наследование. Поэтому расширение класса `Thread` означает, что наш подкласс не сможет наследоваться. Если мы используем `Runnable`-интерфейс, то сможем расширить наш класс.
2. Класс интересен только тем, что может выполняться, и поэтому наследование класса `Thread` будет избыточным.

Синхронизация и блокировки

Потоки в пределах заданного процесса используют пространство памяти совместно. Это имеет свои преимущества и недостатки, поскольку позволяет потокам совместно использовать данные, но приводит к возникновению конфликтов, когда два потока одновременно изменяют ресурс. Для контроля за общими ресурсами Java использует механизм синхронизации.

Ключевые слова `synchronized` и `lock` — основа для реализации синхронного выполнения кода.

Методы синхронизации

Обычно мы ограничиваем доступ к совместно используемым ресурсам с помощью ключевого слова `synchronized`. Оно может применяться к методам и блокам кода, предотвращая одновременный доступ множества потоков к одному и тому же объекту. Для большей ясности 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      lock = new ReentrantLock();
10     public void run() {
11         myObj.foo(name); } (если блокировку вынимать из метода, то
12     } public int withdraw(int value) {           * как же foo() в
13 } lock.lock(); } (также
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 /* Те же ссылки на obj. Только один может вызвать foo,
10 * все остальные будут вынуждены ждать. */
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 на блокировке класса. Два потока из предыдущего кода не могли 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) {
11         /* same as before */
```

продолжение ↗

```

12     } while (distance > 0);
13
14     public static synchronized void bar(String name) {
15         /* same as foo */
16     }
17 }
```

При запуске этого кода вы получите:

```

Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

Синхронизированные блоки кода

Аналогичным образом можно синхронизировать блок кода. Эта операция подобна синхронизации метода.

```

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 }
```

Как и в случае синхронизации метода, только один поток на экземпляр `MyObject` может выполнить код в пределах синхронизируемого блока. Это означает, что если `thread1` и `thread2` работают с общим экземпляром `MyObject`, только один из потоков может выполнить блок кода.

Блокировки

Для более тонкой настройки можно воспользоваться блокировкой. 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.

Наиболее часто блокировка используется, когда к одному ресурсу существует доступ сразу из нескольких мест, но необходимо гарантировать работу только одного потока в конкретный момент времени.

```

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 }
```

Мы добавили код, преднамеренно замедляющий выполнение `withdraw` и `deposit`, поскольку это помогает продемонстрировать потенциальные проблемы. Вам не понадобится писать точно такой же код, но ситуация, которую он отражает, очень реальна. Использование блокировок поможет защитить совместно используемый ресурс от изменения неожиданными способами.

Взаимные блокировки и их предотвращение

Взаимная блокировка — это ситуация, когда поток заблокирован в ожидании ресурса другого потока, который никогда не освободится. Поскольку каждый поток находится в ожидании другого, оба остаются в таком состоянии навсегда. Потоки, как говорится, «зависли».

Чтобы взаимная блокировка произошла, должны быть выполнены все четыре условия:

- Взаимное исключение:** только один процесс получает доступ к ресурсу в установленный период времени. (Или, говоря более корректно, — существует ограниченный доступ к ресурсу. Взаимная блокировка может произойти, если ресурс ограничивает количество подключений.)
- Удержание и ожидание:** удерживающие ресурс процессы могут запросить дополнительные ресурсы, не освобождая при этом текущие.

3. **Отсутствие вытеснения:** один процесс не может насильственно удалить ресурс другого процесса.
4. **Круговое ожидание:** два или более процесса формируют замкнутую цепочку, где каждый процесс ждет 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

- 16.1. В чем разница между потоком и процессом?
- 16.2. Как оценить время, затрачиваемое на контекстное переключение?
- 16.3. В знаменитой задаче об обедающих философах каждый из них имеет только одну палочку для еды. Филосоfu 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 моделирование задачи, предотвращающее взаимные блокировки.
- 16.4. Разработайте класс, обеспечивающий блокировку так, чтобы предотвратить возникновение мертвой блокировки.
- 16.5. Предположим, что у вас есть 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: сначала — ThreadA, потом — ThreadB, потом — ThreadC. Разработайте механизм, гарантирующий, что сначала будет выполнен первый поток, после него — второй, а потом уже — третий.

- 16.6. Дано: класс с синхронизированным методом A и обычным методом C. Если у вас есть два потока в одном экземпляре программы, могут ли они оба выполнить A одновременно? Могут ли они вызвать A и C одновременно?

17. Задачи умеренной сложности

- 17.1.** Напишите функцию, меняющую местами значения переменных, не используя временные переменные.
- 17.2.** Разработайте алгоритм, проверяющий результат игры в крестики-нолики.
- 17.3.** Напишите алгоритм, вычисляющий число конечных нулей в $n!$.
- 17.4.** Напишите метод, находящий максимальное из двух чисел, не используя операторы `if-else` или любые другие операторы сравнения.
- 17.5.** В игру «Гениальный отгадчик¹» играют следующим образом:

У компьютера есть четыре слота, в каждом слоте находится шар красного (R), желтого (Y), зеленого (G) или синего (B) цвета. Последовательность RGGB означает, что в слоте 1 — красный шар, в слотах 2 и 3 — зеленый, 4 — синий. Пользователь должен угадать цвета шаров, например предположить, что там YRGB. Если вы угадали правильный цвет, то вы получаете «хит». Если вы назвали цвет, который присутствует в раскладе, но находится в другом слоте, вы получаете «псевдохит». Слот с «хитом» не может быть назван «псевдохитом». 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».

- 17.6.** Дано: массив целых чисел. Напишите метод, находящий индексы m и n такие, 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)

- 17.7.** Дано: целое число. Выведите его значение прописью (например, «одна тысяча двести тридцать четыре»).

- 17.8.** Дано: массив целых чисел (положительных и отрицательных). Найдите непрерывную последовательность с самой большой суммой. Возвратите сумму.

Пример

Ввод: 2, -8, 3, -2, 4, -10

Вывод: 5 (соответствующая последовательность: 3, -2, 4)

¹ Mastermind — вариант игры «Быки и коровы». — Примеч. ред.

17.9. Разработайте метод, вычисляющий частоту использования заданного слова в книге.

17.10. Поскольку XML излишне «многословен», закодируйте его так, чтобы каждый тег преобразовывался в какое-либо целое значение:

```

Element --> Tag Attributes END Children END
Attribute --> Tag Value
END --> 0
Tag --> любое предопределенное целое значение
Value --> string value END

```

Преобразуем XML-код в сжатую форму, подразумевая, 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**).

17.11. Реализуйте метод **rand7()** на базе метода **rand5()**. Другими словами, имеется метод, генерирующий случайные числа в диапазоне от 0 до 4 (включительно). Напишите использующий его метод, генерирующий случайное число в диапазоне от 0 до 6 (включительно).

17.12. Разработайте алгоритм, обнаруживающий в массиве все пары целых чисел, сумма которых равна заданному значению.

17.13. Дано: простая графообразная структура данных **BiNode**, содержащая указатели на два других узла:

```

1 public class BiNode {
2     public BiNode node1, node2;
3     public int data;
4 }

```

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 работа алгоритма должна осуществляться в исходной структуре данных.

17.14. О, нет! Вы только что закончили длинный документ, но сделали неудачную операцию Поиск/Замена. Вы случайно удалили все пробелы, пунктуацию и все прописные буквы. Предложение **I reset the computer. It still didn't boot!** превратилось в **iresetthecomputeritstilldidntboot**. После того как вы разделите строку на отдельные слова, можно будет восстановить пунктуацию. Большинство слов, кроме имен собственных, находятся в словаре.

Для заданного словаря (списка слов) разработайте алгоритм, выполняющий оптимальную деконкатенацию строки. Алгоритм должен минимизировать число нераспознанных последовательностей символов.

Строку "jesslookedjustliketimherbrother" необходимо оптимальным образом преобразовать в "JESS looked just like TIM her brother". Парсер не справился с семью символами, которые были написаны в верхнем регистре.

Максимально возможное количество символов в строке равно n , а длина каждого слова в словаре не превышает k . Каждый символ строки может быть либо частью слова, либо независимым от него.

Слово

18.4. Для заданного словаря S и строки A определите минимальное количество символов, необходимое для преобразования A в строку из слов из S .

Упражнение

Бытовой

Задача

Бытовой

Задача

18.5. Для заданного текстового файла, содержащего слова, определите количество максимальных зонагрупп (абзацекомбинаций) между строками, в которых каждая строка содержит не более одного слова. Постройте алгоритм $O(1)$.

Комплексный

18.6. Определите минимальное количество символов, необходимое для преобразования строки A в строку из слов из S , если для каждого слова в S известны минимальные и максимальные длины.

Одноточечный

Задача

18.7. Для заданного списка слов S определите минимальное количество символов, необходимое для преобразования строки A в строку из слов из S , если для каждого слова в S известны минимальные и максимальные длины.

Упражнение

Бытовой

Задача

Бытовой

18.8. Дана строка A и массив минимум слов T . Проверьте методом ℓ -последовательностей, что для каждого слова в T найдется подстрока A , соответствующая ему.

Комплексный

Задача

18.9. Определите минимальное количество символов, необходимое для преобразования строки A в строку из слов из S , если для каждого слова в S известны минимальные и максимальные длины.

Комплексный

Задача

Комплексный

18.10. Для заданного набора слов S определите минимальное количество символов, необходимое для преобразования строки A в строку из слов из S , если для каждого слова в S известны минимальные и максимальные длины.

Упражнение

Бытовой

Задача

Бытовой

Пример: DAWW < LWWB < LWWB < LWWB < LWWB < LKE

18. Задачи повышенной сложности

-
- 18.1. Напишите функцию суммирования двух чисел без использования «+» и других арифметических операторов.
 - 18.2. Напишите метод, тасующий карточную колоду. Колода должна быть идеально перемешана. Перестановки карт должны быть равновероятными. (Вы можете использовать идеальный генератор случайных чисел.)
 - 18.3. Напишите метод, генерирующий случайную последовательность m целых чисел из массива размером n . Все элементы выбираются с одинаковой вероятностью.
 - 18.4. Напишите метод, который будет подсчитывать количество цифр «2», используемых 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.)

-
- 18.5. Дано: большой текстовый файл, содержащий слова. Напишите код, который позволяет найти минимальное расстояние (выражаемое количеством слов) между любыми двумя словами в файле. Достаточно ли будет $O(1)$ времени? Сколько памяти понадобится для решения?
 - 18.6. Опишите алгоритм для нахождения миллиона наименьших чисел в наборе из миллиарда чисел. Память компьютера позволяет хранить весь миллиард чисел.
 - 18.7. 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

-
- 18.8. Дано: строка s и массив меньших строк T . Разработайте метод поиска s для каждой меньшей строки в T .
 - 18.9. Сгенерированные случайным образом числа передаются методу. Напишите программу расчета среднего значения, динамически отслеживающую поступающие новые значения.
 - 18.10. Для двух слов одинаковой длины напишите метод, трансформирующий одно слово в другое, изменяя одну букву за один шаг. Каждое новое слово должно присутствовать в словаре.

Пример

Ввод: DAMP, LIKE

Вывод: DAMP -> LAMP -> LIMP -> LIME -> LIKE

- 18.11.** Представьте, что существует квадратная матрица, каждый пиксель которой может быть черным или белым. Разработайте алгоритм поиска максимального субквадрата, у которого все стороны черные.
- 18.12.** Дано: матрица размером $N \times N$, содержащая положительные и отрицательные числа. Напишите код поиска субматрицы с максимально возможной суммой.
- 18.13.** Дано: список из миллиона 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.

1. Используйте алгоритм dynamic programming для решения задачи о максимальном неподsecutive подстроке в списке из n строк. Для этого можно использовать таблицу длины подстроки для каждого символа в каждой строке.

Сложность каждого символа в строке будет $O(n)$, а общая сложность алгоритма $O(n^2)$.
Однако если использовать алгоритм dynamic programming для каждого символа в строке, то сложность будет $O(n^3)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Для решения задачи о максимальном неподsecutive подстроке в списке из n строк можно использовать алгоритм dynamic programming для каждого символа в строке, но сложность алгоритма будет $O(n^2)$.

Часть IX Решения

Присоединяйтесь к нам на www.CrackingTheCodingInterview.com и загружайте полные, совместимые с Java/Eclipse решения, читайте обсуждения задач из этой книги с другими пользователями, отправляйте нам свои отзывы, просматривайте список ошибок в этой книге, отправляйте свое резюме и обращайтесь за дополнительным советом.

СТРУКТУРЫ ДАННЫХ: РЕШЕНИЯ

1. Массивы и строки

- 1.1.** Реализуйте алгоритм, определяющий, все ли символы в строке встречаются один раз. При выполнении этого задания нельзя использовать дополнительные структуры данных.

Решение

Сразу уточните у интервьюера, является ли строка обычной ASCII-строкой или это строка Unicode. Этот вопрос продемонстрирует ваше внимание к деталям и глубокое понимание информатики. Предположим, что используется набор символов ASCII. Если это не так, то нам нужно будет увеличить объем памяти для хранения строки, но логика решения останется той же.

Можно слегка оптимизировать нашу задачу — возвращать `false`, если длина строки превышает количество символов в алфавите. В конце концов, не может существовать строки с 280 уникальными символами, если символов всего 256.

Наше решение заключается в создании массива логических значений, где флаг с индексом i означает, содержится ли символ алфавита i в строке. Если вы «наткнетесь» на этот же символ во второй раз, можете сразу возвращать `false`.

Код, реализующий этот алгоритм, представлен ниже:

```
1 public boolean isUniqueChars2(String str) {  
2     if (str.length() > 256) return false;  
3  
4     boolean[] char_set = new boolean[256];  
5     for (int i = 0; i < str.length(); i++) {  
6         int val = str.charAt(i);  
7         if (char_set[val]) {           // Символ уже был найден в строке  
8             return false;  
9         }  
10        char_set[val] = true;  
11    }  
12    return true;  
13 }
```

Оценка времени выполнения этого кода — $O(n)$, где n — длина строки, оценка требуемого пространства — $O(1)$.

Можно уменьшить использование памяти за счет битового вектора. В следующем коде мы предполагаем, что в строке есть только символы в нижнем регистре a-z. Это позволит нам использовать просто одно значение типа int.

```

1 public boolean isUniqueChars(String str) {
2     if (str.length() > 256) return false;
3
4     int checker = 0;
5     for (int i = 0; i < str.length(); i++) {
6         int val = str.charAt(i) - 'a';
7         if ((checker & (1 << val)) > 0) {
8             return false;
9         }
10        checker |= (1 << val);
11    }
12    return true;
13 }
```

Можно использовать другой подход:

- Сравнить каждый символ строки с любым другим символом строки. Это потребует $O(n^2)$ времени и $O(1)$ памяти.
- 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 осторожны: некоторые алгоритмы сортировки требуют больших объемов памяти.

Эти решения не оптимальны, но лучше соответствуют дополнительным ограничениям конкретной задачи.

1.2. Реализуйте функцию void reverse(char* str) на C или C++, функция должна циклически сдвигать строку, заканчивающуюся символом null.

Решение

Это классический вопрос. Единственный «глюк», который может произойти, — это неосторожное обращение с символом null.

Реализуем эту функцию на C.

```

1 void reverse(char *str) {
2     char* end = str;
3     char tmp;
4     if (str) {
5         while (*end) { /* находим конец строки */
6             ++end;
7         }
8         --end; /* на один символ назад, ведь последний символ строки - null */
9
10        /* меняем символы местами от начала строки до ее конца, пока указатели
не встретятся в середине */
11 }
```

```

12         while (str < end) {
13             tmp = *str;
14             *str++ = *end;
15             *end-- = tmp;
16         }
17     }
18 }
```

Существует много способов решения этой задачи. Можно реализовать данный код рекурсивно, но лучше этого не делать.

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

Решение

Сразу уточните детали у интервьюера. Следует разобраться, является ли сравнение анаграмм чувствительным к регистру. То есть является ли строка "God" анаграммой "дог"? Дополнительно нужно выяснить, учитываются ли пробелы.

Предположим, что для данной задачи регистр символов учитывается, а пробелы являются существенными. Поэтому строки "dog" и "dog" не совпадают.

Сравнивая две строки, помните, что строки разной длины не могут быть анаграммами.

Существует два простых способа решить эту задачу.

Способ 1. Сортировка строк

Если строки являются анаграммами, то они состоят из одинаковых символов, расположенных в разном порядке. Сортировка двух строк должна упорядочить символы. Теперь остается только сравнить две отсортированные версии строк.

```

1 public String sort(String s) {
2     char[] content = s.toCharArray();
3     java.util.Arrays.sort(content);
4     return new String(content);
5 }
6
7 public 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 нельзя назвать оптимальным во всех смыслах, он удачен, поскольку его легко понять. С практической точки зрения это превосходный способ решить задачу. Однако если важна эффективность, придется реализовать другой алгоритм.

Способ 2. Проверка счетчиков идентичных символов

Для реализации этого алгоритма можно использовать свойство анаграммы — одинаковые «счетчики» символов. Мы просто подсчитываем, сколько раз встречался

каждый символ в строке. Затем сравниванием массивы, полученные для каждой строки.

```

1 public boolean permutation(String s, String t) {
2     if (s.length() != t.length()) {
3         return false;
4     }
5
6     int[] letters = new int[256]; // предположение
7
8     char[] s_array = s.toCharArray();
9     for (char c : s_array) { // подсчитываем количество 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         if (--letters[c] < 0) {
16             return false;
17         }
18     }
19
20     return true;
21 }
```

Обратите внимание на строку 6. На собеседовании вы всегда должны согласовывать с интервьюером размер алфавита. (Здесь используется набор символов ASCII.)

- 1.4.** Напишите метод, заменяющий все пробелы в строке символами '%20'. Можно предположить, что длина строки позволяет сохранить дополнительные символы и «истинная» длина строки известна. (Примечание: при реализации метода на Java используйте символьный массив.)

Решение

Общий подход в задачах обработки строк — двигаться от конца строки к началу. Это полезно, потому что у нас есть дополнительное пространство в конце строки, которое позволяет изменять символы, не думая о том, что информация потеряется.

Давайте используем этот подход в данной задаче. Алгоритм использует двупроходное сканирование. При первом проходе мы подсчитываем, сколько пробелов находится в строке. Это позволяет расчитать длину результирующей строки. Когда мы находим пробел, подставляем вместо него %20. Если символ не является пробелом, он просто копируется.

Код, реализующий данный алгоритм:

```

1 public void replaceSpaces(char[] str, int length) {
2     int spaceCount = 0, newLength, i = 0;
3     for (i = 0; i < length; i++) {
4         if (str[i] == ' ') {
5             spaceCount++;
6         }
7     }
```

продолжение ↗

```

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 }

```

Мы решили эту задачу, используя символьные массивы, так как строки в Java являются неизменными. Если использовать непосредственно строки, это потребует возврата новой копии строки, но позволит решить задачу за один проход.

- 1.5.** Реализуйте метод, осуществляющий сжатие строки, на основе счетчика повторяющихся символов. Например, строка `aabccccc` должна превратиться в `a2b1c5a3`. Если «сжатая» строка оказывается длиннее исходной, метод должен вернуть исходную строку.

Решение

На первый взгляд реализация этого метода кажется довольно прямолинейной. Мы выполняем итерации через строку, копируя символы в новую строку и подсчитывая повторы.

```

1 public String compressBad(String str) {
2     String mystr = "";
3     char last = str.charAt(0);
4     int count = 1;
5     for (int i = 1; i < str.length(); i++) {
6         if (str.charAt(i) == last) { // Находим повторяющийся символ
7             count++;
8         } else { // Вставляем счетчик символа и обновляем последний символ
9             mystr += last + "" + count;
10            last = str.charAt(i);
11            count = 1;
12        }
13    }
14    return mystr + last + count;
15 }

```

Этот код не отслеживает случай, когда сжатая строка получается длиннее исходной. Но эффективен ли этот алгоритм?

Давайте оценим время выполнения этого кода: $O(p + k^2)$, где p — размер исходной строки, а k — количество последовательностей символов. Например, если строка `aabccdeeeaa` содержит 6 последовательностей символов. Алгоритм работает медлен-

но, поскольку используется конкатенация строк, требующая $O(n^2)$ времени (см. `StringBuffer` в гл. 1).

Улучшить код можно, используя `StringBuffer`:

```

1 String compressBetter(String str) {
2     /* Проверяем, вдруг сжатие создаст более длинную строку */
3     int size = countCompression(str);
4     if (size >= str.length()) { // Иногда лучше не сжимать
5         return str;
6     }
7
8     StringBuffer mystr = new StringBuffer();
9     char last = str.charAt(0);
10    int count = 1;
11    for (int i = 1; i < str.length(); i++) {
12        if (str.charAt(i) == last) { // Найден повторяющийся символ
13            count++;
14        } else { // Вставляем счетчик символов, обновляем последний символ
15            mystr.append(last); // Вставляем символ
16            mystr.append(count); // Вставляем счетчик
17            last = str.charAt(i);
18            count = 1;
19        }
20    }
21
22    /* В строках 15-16 символы вставляются, когда
23     * изменяется повторяющийся символ. Мы должны обновить строку
24     * в конце метода, так как самый последний повторяющийся символ
25     * еще не был установлен в сжатой строке
26     */
27    mystr.append(last);
28    mystr.append(count);
29    return mystr.toString();
30 }
31
32 int countCompression(String str) {
33     char last = str.charAt(0);
34     int size = 0;
35     int count = 1;
36     for (int i = 1; i < str.length(); i++) {
37         if (str.charAt(i) == last) { // Каждый раз, когда мы видим одинаковую
38             count++; // строку, увеличиваем счетчик
39         } else {
40             last = str.charAt(i);
41             size += 1 + String.valueOf(count).length();
42             count = 0;
43         }
44     }
45     size += 1 + String.valueOf(count).length();
46     return size;
47 }
```

Этот алгоритм намного эффективнее. Обратите внимание на проверку размера в строках 2–5.

Если мы не хотим (или не можем) использовать `StringBuffer`, можно решить эту задачу иначе. В строке 2 рассчитывается конечный размер строки, что позволит создать массив подходящего размера:

```

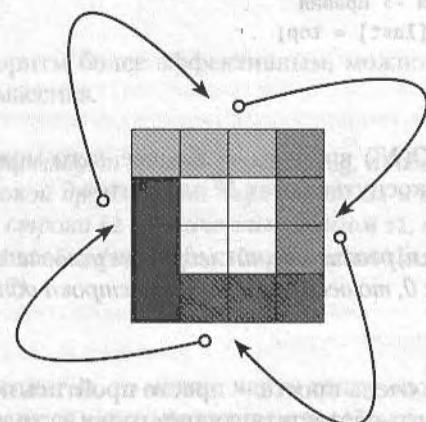
1 String compressAlternate(String str) {
2     /* Проверяем, вдруг сжатие создаст более длинную строку */
3     int size = countCompression(str);
4     if (size >= str.length()) {
5         return str;
6     }
7
8     char[] array = new char[size];
9     int index = 0;
10    char last = str.charAt(0);
11    int count = 1;
12    for (int i = 1; i < str.length(); i++) {
13        if (str.charAt(i) == last) { // Найдите повторяющийся символ
14            count++;
15        } else {
16            /* Обновляем счетчик повторяющихся символов */
17            index = setChar(str, array, last, index, count);
18            last = str.charAt(i);
19            count = 1;
20        }
21    }
22
23    /* Обновляем строку с последним набором повторяющихся символов */
24    index = setChar(str, array, last, index, count);
25    return String.valueOf(array);
26 }
27
28 int setChar(String str, char[] array, char c, int index,
29             int count) {
30     array[index] = c;
31     index++;
32
33     /* Конвертируем счетчик в строку */
34     char[] cnt = String.valueOf(count).toCharArray();
35
36     /* Копируем символы от большего разряда к меньшему */
37     for (char x : cnt) {
38         array[index] = x;
39         index++;
40     }
41     return index;
42 }
43
44 int countCompression(String str) {
45     /* так же, как и раньше */
46 }
```

Подобно предыдущему решению, этот код потребует $O(N)$ времени и $O(N)$ пространства.

- 1.6.** Дано: изображение в виде матрицы размером $N \times N$, где каждый пиксель занимает 4 байта. Напишите метод, поворачивающий изображение на 90° .

Решение

Поскольку мы будем поворачивать матрицу на 90° , простейший способ — использовать слои. Мы произведем циклическое вращение слоя, двигаясь по кругу.



Как поменять местами четырьмя сторонами? Один из вариантов — скопировать верхнюю сторону в массив, затем переместить левую сторону на место верхней, нижнюю — на место левой и т. д. На это потребуется $O(N)$ памяти, которая фактически является избыточной.

Лучший способ сделать то же самое — реализовать последовательную замену элемента за элементом:

```
1 for i = 0 to n
2 temp = top[i];
3 top[i] = left[i]
4 left[i] = bottom[i]
5 bottom[i] = right[i]
6 right[i] = temp
```

Замена делается на каждом слое начиная с наиболее удаленного. (Альтернативное решение — двигаться от внутреннего слоя к внешнему.)

Код этого алгоритма:

```
1 public void rotate(int[][] matrix, int n) {
2     for (int layer = 0; layer < n / 2; ++layer) {
3         int first = layer;
4         int last = n - 1 - layer;
5         for (int i = first; i < last; ++i) {
6             int offset = i - first;
7             // сохраняем вершину
8             int top = matrix[first][i];
```

продолжение ↗

```

9
10         // левая -> верхняя
11         matrix[first][i] = matrix[last-offset][first];
12
13         // нижняя -> левая
14         matrix[last-offset][first] = matrix[last][last - offset];
15
16         // правая -> нижняя
17         matrix[last][last - offset] = matrix[i][last];
18
19         // верхняя -> правая
20         matrix[i][last] = top;
21     }
22 }
23 }
```

Этот алгоритм требует $O(N^2)$ времени — лучшее, чего можно добиться, поскольку любой алгоритм должен коснуться всех N^2 элементов.

1.7. Напишите алгоритм, реализующий следующее условие: если элемент матрицы в точке $M \times N$ 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 обнуляется. Один из способов — создать вторую матрицу, содержащую флаги «исходных» нулей. Но тогда потребуется сделать два прохода по матрице, что потребует $O(NM)$ пространства.

Так ли нам нужно $O(NM)$? Нет. Так как мы собираемся обнулять строки и столбцы, нет необходимости запоминать значения этих элементов. Пусть ноль находится в ячейке [2][4]. Это означает, что необходимо обнулить строку 2 и столбец 4. А если мы обнуляем эти строку и столбец, то зачем их запоминать?

Приведенный ниже код реализует наш алгоритм. Мы используем два массива, чтобы отследить все строчки и столбцы с нулями. После чего делаем второй проход и расставляем нули на основании созданного массива.

```

1 public void setZeros(int[][] matrix) {
2     boolean[] row = new boolean[matrix.length];
3     boolean[] column = new boolean[matrix[0].length];
4
5     // Сохраняем индекс ряда и колонки со значением 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    for (int i = 0; i < matrix.length; i++) {
16        for (int j = 0; j < matrix[0].length; j++) {
17            if (row[i] || column[j]) {
18                matrix[i][j] = 0;
19            }
20        }
21    }
22 }
```

```

12     }
13 }
14
15 // Устанавливаем arr[i][j] в 0, если в ряде i или колонке j есть 0
16 for (int i = 0; i < matrix.length; i++) {
17     for (int j = 0; j < matrix[0].length; j++) {
18         if (row[i] || column[j]) {
19             matrix[i][j] = 0;
20         }
21     }
22 }
23 }

```

Чтобы сделать наш алгоритм более эффективным, можно использовать двоичный вектор вместо булевого массива.

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

Решение

Если `s2` — циклический сдвиг `s1`, то можно найти точку «поворота». Например, если вы сдвигаете `waterbottle` после `wat`, то получите `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`.

Чтобы решить задачу, достаточно вызвать функцию `isSubstring(s1s1, s2)`.

Следующий код реализует описанный алгоритм:

```

1 public boolean isRotation(String s1, String s2) {
2     int len = s1.length();
3     /* проверяем, что у s1 и s2 одинаковая длина и они не пусты */
4     if (len == s2.length() && len > 0) {
5         /* конкатенация s1 и s1 */
6         String s1s1 = s1 + s1;
7         return isSubstring(s1s1, s2);
8     }
9     return false;
10 }

```

2. Связные списки

- 2.1.** Напишите код, удаляющий дубликаты из несортированного связного списка.

Дополнительно

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

Решение

Чтобы удалить копии из связного списка, их нужно сначала найти. Для этого подойдет простая хэш-таблица. В приведенном далее решении выполняется проход по списку, каждый элемент которого добавляется в хэш-таблицу. Когда обнаруживается повторяющийся элемент, он удаляется, и цикл продолжает работу. За счет использования связного списка всю задачу можно решить за один проход.

```

1 public static void deleteDups(LinkedListNode n) {
2     Hashtable table = new Hashtable();
3     LinkedListNode previous = null;
4     while (n != null) {
5         if (table.containsKey(n.data)) {
6             previous.next = n.next;
7         } else {
8             table.put(n.data, true);
9             previous = n;
10        }
11        n = n.next;
12    }
13 }
```

Приведенное решение потребует $O(N^2)$ времени, где N — количество элементов в связном списке.

Дополнительное ограничение: использование буфера запрещено.

В этом случае мы можем реализовать цикл с помощью двух указателей: `current` (работает через связный список) и `runner` (проверяет все последующие узлы на наличие дубликатов).

```

1 public static void deleteDups(LinkedListNode head) {
2     if (head == null) return;
3
4     LinkedListNode current = head;
5     while (current != null) {
6         /* Удаляем все следующие узлы с таким же значением */
7         LinkedListNode runner = current;
8         while (runner.next != null) {
9             if (runner.next.data == current.data) {
10                 runner.next = runner.next.next;
11             } else {
12             }
13         }
14     }
15 }
```

```

12         runner = runner.next;           // Операция с головой
13     }
14 }
15 current = current.next;
16 }
17 }
```

Данный код требует всего $O(1)$ пространства, но занимает $O(N^2)$ времени.

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

Решение

Данную задачу можно решить рекурсивным и нерекурсивным способом. Рекурсивные решения обычно более понятны, но менее оптимальны. Например, рекурсивная реализация этой задачи почти в два раза короче нерекурсивной, но занимает $O(n)$ пространства, где n — количество элементов связного списка.

При решении данной задачи помните, что можно выбрать значение k так, что при передаче $k = 1$ мы получим последний элемент, 2 — предпоследний и т. д. Или выбрать значение 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 ($\text{длина} - k$). Нужно пройтись по списку и найти этот элемент. Поскольку такое решение тривиально, то скорее всего, оно наименее удовлетворит интервьюера.

Решение 2. Рекурсивное решение

Такой алгоритм рекурсивно проходит связный список. По достижении последнего элемента алгоритм начинает обратный отсчет, и счетчик сбрасывается в 0. Каждый шаг инкрементирует счетчик на 1. Когда счетчик достигнет k , искомый элемент будет найден.

Реализация этого алгоритма коротка и проста — достаточно «передать назад» целое значение через стек. К сожалению, оператор `return` не может вернуть значение узла. Так как же обойти эту трудность?

Подход А: не возвращайте элемент

Можно не возвращать элемент, достаточно вывести его сразу, как только он будет найден. А в операторе `return` вернуть значение счетчика.

```

1 public static int nthToLast(ListNode head, int k) {
2     if (head == null) {
3         return 0;
4     }
5     int i = nthToLast(head.next, k) + 1;
6     if (i == k) {
7         System.out.println(head.data);
8     }
9     return i;
10 }
```

Конечно, данное решение правильно, но достаточно ли этого интервьюеру?

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 путем передачи указателя на него.

```

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 }
```

Подход В: создайте интерфейсный класс

Как было описано ранее, проблема заключается в том, что мы не можем одновременно передать счетчик и индекс. Если мы «обернем» значение counter в специальный класс (или просто элемент массива), то сможем передать ссылку.

```

1 public class IntWrapper {
2     public int value = 0;
3 }
4
5 LinkedListNode nthToLastR2(LinkedListNode head, int k,
6                             IntWrapper i) {
7     if (head == null) {
8         return null;
9     }
10    LinkedListNode node = nthToLastR2(head.next, n, i);
11    i.value = i.value + 1;
12    if (i.value == k) { // Мы нашли k-й элемент с конца
13        return head;
14    }
15    return node;
16 }
```

Каждое из приведенных решений занимает $O(n)$ пространства из-за рекурсивных вызовов.

Существует множество разнообразных решений. Можно сохранить счетчик в статической переменной. Или создать класс, который хранит и 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. Итерационное решение

Итерационное решение будет более сложным, но и более оптимальным. Можно использовать два указателя — p1 и p2. Сначала оба указателя указывают на начало списка. Затем мы перемещаем p2 на k узлов вперед. Теперь мы начинаем перемещать

оба указателя одновременно. Когда p_2 дойдет до конца списка, p_1 будет указывать на нужный нам элемент.

Следующий код реализует данный алгоритм:

```

1 LinkedListNode nthToLast(LinkedListNode head, int k) {
2     if (k <= 0) return null;
3
4     LinkedListNode p1 = head;
5     LinkedListNode p2 = head;
6
7     // Перемещаем p2 вперед на k узлов
8     for (int i = 0; i < k - 1; i++) {
9         if (p2 == null) return null; // Ошибка
10        p2 = p2.next;
11    }
12    if (p2 == null) return null;
13
14    /* Теперь перемещаем p1 и p2 с одинаковой скоростью. Когда p2 дойдет до конца,
15     * p1 будет над нужным элементом. */
16    while (p2.next != null) {
17        p1 = p1.next;
18        p2 = p2.next;
19    }
20    return p1;
21 }
```

Этот алгоритм требует $O(n)$ времени и $O(1)$ пространства.

2.3. Реализуйте алгоритм, удаляющий узел из середины односвязного списка (доступ дан только к этому узлу).

Решение

В этой задаче вы не имеете доступа к началу списка, в вашем распоряжении только доступ к конкретному узлу. Для решения этой задачи вам нужно просто скопировать данные из следующего узла в текущий, а затем удалить следующий узел.

Следующий код реализует этот алгоритм:

```

1 public static boolean deleteNode(LinkedListNode n) {
2     if (n == null || n.next == null) {
3         return false; // Failure
4     }
5     LinkedListNode next = n.next;
6     n.data = next.data;
7     n.next = next.next;
8     return true;
9 }
```

Обратите внимание, что задача не может быть решена, если удаляемый узел является последним узлом связного списка. Ничего страшного, просто интервьюер хочет, чтобы вы обсудили с ним, что делать в этом случае. Можно, например, пометить данный узел как пустой.

- 2.4. Напишите код, разбивающий связный список вокруг значения x так, чтобы все узлы, меньшие x , оказались перед узлами, большими или равными x .

Решение

Если бы мы работали с массивом, то было бы много сложностей, связанных со смещением элементов.

Со связным списком задача намного проще. Вместо того чтобы смещать 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, расставляя элементы по спискам `before` и `after`. Как только конец исходного связного списка будет достигнут, можно выполнить слияние получившихся списков.

Приведенный код реализует данный подход:

```

1 /* Передаем начало списка, который нужно разделить, и значение x, вокруг которого
2 * список будет разделен */
3 public ListNode partition(ListNode node, int x) {
4     ListNode beforeStart = null;
5     ListNode beforeEnd = null;
6     ListNode afterStart = null;
7     ListNode afterEnd = null;
8
9     /* Разбиваем список */
10    while (node != null) {
11        ListNode next = node.next;
12        node.next = null;
13        if (node.data < x) {
14            /* Вставляем узел в конец списка before */
15            if (beforeStart == null) {
16                beforeStart = node;
17                beforeEnd = beforeStart;
18            } else {
19                beforeEnd.next = node;
20                beforeEnd = node;
21            }
22        } else {
23            /* Вставляем узел в конец списка after */
24            if (afterStart == null) {
25                afterStart = node;
26                afterEnd = afterStart;
27            } else {
28                afterEnd.next = node;
29                afterEnd = node;
30            }
31        }
32        node = next;
33    }
34
35    if (beforeStart == null) {
```

```

36     return afterStart;
37 }
38 /* Слияние списков before и after */
39 beforeEnd.next = afterStart;
40 return beforeStart;
41 }
42 }
```

Если вы не хотите использовать четыре переменные, чтобы отслеживать всего два связных списка, можно избавиться от части из них за счет небольшой потери эффективности. Но «ущерб» будет не очень велик, поэтому время «большого О» останется тем же самым, зато код станет более коротким и красивым.

Альтернативное решение: вместо вставки узлов в конец списков `before` и `after` можно вставлять элементы в начало списка.

```

1 public LinkedListNode partition(LinkedListNode node, int x) {
2     LinkedListNode beforeStart = null;
3     LinkedListNode afterStart = null;
4
5     /*Разбиваем список */
6     while (node != null) {
7         LinkedListNode next = node.next;
8         if (node.data < x) {
9             /* Вставляем узел в начало списка before */
10            node.next = beforeStart;
11            beforeStart = node;
12        } else {
13            /* Вставляем узел в начало списка after */
14            node.next = afterStart;
15            afterStart = node;
16        }
17        node = next;
18    }
19
20    /* Выполняем слияние списков */
21    if (beforeStart == null) {
22        return afterStart;
23    }
24
25    /* Находим конец списка before и соединяем списки*/
26    LinkedListNode head = beforeStart;
27    while (beforeStart.next != null) {
28        beforeStart = beforeStart.next;
29    }
30    beforeStart.next = afterStart;
31
32    return head;
33 }
```

Обратите внимание на нулевые значения. В строке 7 добавлена дополнительная проверка. Необходимо сохранить следующий узел во временной переменной так, чтобы запомнить, какой узел будет следующим.

- 2.5.** Два числа хранятся в виде связных списков, где каждый узел содержит один разряд. Все цифры хранятся в обратном порядке, при этом первая цифра числа находится в начале списка. Напишите функцию, которая суммирует два числа и возвращает результат в виде связного списка.

Дополнительно:

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

Решение

Полезно помнить алгоритм, по которому осуществляется суммирование. Рассмотрим задачу:

$$\begin{array}{r} 6 \ 1 \ 7 \\ + 2 \ 9 \ 5 \\ \hline \end{array}$$

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 реализовать 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:

$$\begin{array}{r} 7 \rightarrow 1 \rightarrow 6 \\ + 5 \rightarrow 9 \rightarrow 2 \\ \hline \end{array}$$

Нам нужно сделать следующее:

Сложить 7 и 5, получить 12. Двойка становится первым узлом нашего связного списка. Единица «запоминается» для следующей суммы:

List: 2 → ?

Затем мы суммируем 1 и 9, а также запомненную единицу. Получаем результат 11. Единица становится вторым элементом, а другая единица запоминается для следующей суммы:

List: 2 → 1 → ?

Затем мы добавляем 6, 2 и запомненную единицу, получаем 9 — последний элемент нашего списка.

List: 2 → 1 → 9.

Следующий код реализует описанный алгоритм:

```

1 ListNode addLists(ListNode l1, ListNode l2,
2                     int carry) {
3     /* Все готово — оба списка null и запомненное значение (carry — перенос) = 0 */
4     if (l1 == null && l2 == null && carry == 0) {
5         return null;
6     }
7
8     ListNode result = new ListNode(carry, null, null);
9
10    /* Добавляем значение и дату из l1 и l2 */
11    int value = carry;

```

```

12     if (l1 != null) {
13         value += l1.data;
14     }
15     if (l2 != null) {
16         value += l2.data;
17     }
18
19     result.data = value % 10; /* Второй разряд числа */
20
21     /* Рекурсия */
22     if (l1 != null || l2 != null || value >= 10) {
23         LinkedListNode more = addLists(l1 == null ? null : l1.next,
24                                         l2 == null ? null : l2.next,
25                                         value >= 10 ? 1 : 0);
26         result.setNext(more);
27     }
28     return result;
29 }
```

В этом коде необходимо предусмотреть ситуацию, когда один список короче другого. Ведь мы не хотим получить ошибку с нулевым указателем!

Дополнительное условие

Алгоритм остается практически таким же (рекурсия, перенос), но с небольшими «усложнениями» в реализации.

Один список может оказаться короче другого, а мы не можем обработать это «на лету». Например, представьте, что вы суммируете списки $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4)$ и $(5 \rightarrow 6 \rightarrow 7)$. Нам нужно знать, что 5 должно «совпадать» с 2, а не с 1. Мы можем реализовать это через сравнение длин списков, добавив в начало более короткого списка нули.

В первом алгоритме результаты добавлялись в хвост списка. Это означает, что рекурсивный вызов должен передать `carry` и возвратить результат (который добавляется в хвост). Но в этом случае результат добавляется в начало списка. Это несложно, но решение становится более громоздким. Избавиться от этой проблемы позволит интерфейсный класс `PartialSum`.

Следующий код реализует наш алгоритм:

```

1 public class PartialSum {
2     public ListNode sum = null;
3     public int carry = 0;
4 }
5
6 ListNode addLists(ListNode l1, ListNode l2) {
7     int len1 = length(l1);
8     int len2 = length(l2);
9
10    /* Дополняем более короткий список нулями */
11    if (len1 < len2) {
12        l1 = padList(l1, len2 - len1);
13    } else {
14        l2 = padList(l2, len1 - len2);
```

} (лист 15 продолжение ↗

```

15     }
16
17     /* Сложение списков */
18     PartialSum sum = addListsHelper(l1, l2);
19
20     /* если есть значение переноса, вставьте его в начало списка
21     * в противном случае просто возвратите список. */
22     if (sum.carry == 0) {
23         return sum.sum;
24     } else {
25         LinkedListNode result = insertBefore(sum.sum, sum.carry);
26         return result;
27     }
28 }
29
30 PartialSum addListsHelper(LinkedListNode l1, LinkedListNode l2) {
31     if (l1 == null & l2 == null) {
32         PartialSum sum = new PartialSum();
33         return sum;
34     }
35     /* Добавляем меньшие разряды рекурсивно */
36     PartialSum sum = addListsHelper(l1.next, l2.next);
37
38     /* Добавляем перенос к текущим данным */
39     int val = sum.carry + l1.data + l2.data;
40
41     /* Вставляем сумму текущих разрядов */
42     LinkedListNode full_result = insertBefore(sum.sum, val % 10);
43
44     /* Возвращаем сумму */
45     sum.sum = full_result;
46     sum.carry = val / 10;
47     return sum;
48 }
49
50 /* Заполняем список нулями */
51 LinkedListNode padList(LinkedListNode l, int padding) {
52     LinkedListNode head = l;
53     for (int i = 0; i < padding; i++) {
54         LinkedListNode n = new LinkedListNode(0, null, null);
55         head.prev = n;
56         n.next = head;
57         head = n;
58     }
59     return head;
60 }
61
62 /* Helper function to insert node in the front of a linked list */
63 LinkedListNode insertBefore(LinkedListNode list, int data) {
64     LinkedListNode node = new LinkedListNode(data, null, null);
65     if (list != null) {

```

```

66     list.prev = node;
67     node.next = list;
68   }
69   return node;
70 }

```

Обратите внимание, что `insertBefore()`, `padList()` и `length` (листигн не приводится) вынесены в отдельные методы. Это делает код более понятным и удобным для чтения, что немаловажно для собеседования.

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

Решение

Эта задача является разновидностью классической задачи, задаваемой на собеседованиях, — определить, содержит ли связный список петлю. Давайте используем подход «Сопоставление с образцом».

Часть 1. Определяем, есть ли в связном списке петля

Простейший способ выяснить, есть ли в связном списке петля, — использовать метод бегунка (быстрый/медленный). `FastRunner` делает два шага за один такт, а `SlowRunner` — только один. Подобно двум гоночным автомобилям, мчащимся по одной трассе разными путями, они непременно встретиться.

Проницательный читатель может задать вопрос: может ли быстрый бегунок «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 `SlowRunner` был в точке $i-1$, а `FastRunner` — $((i+1)-2)=i-1$. Следовательно, столкновение неизбежно.

Часть 2. Когда же они встретятся?

Давайте введем обозначение: k — длина связного списка в разомкнутом виде. Как узнать, когда `FastRunner` и `SlowRunner` встретятся, используя алгоритм из части 1?

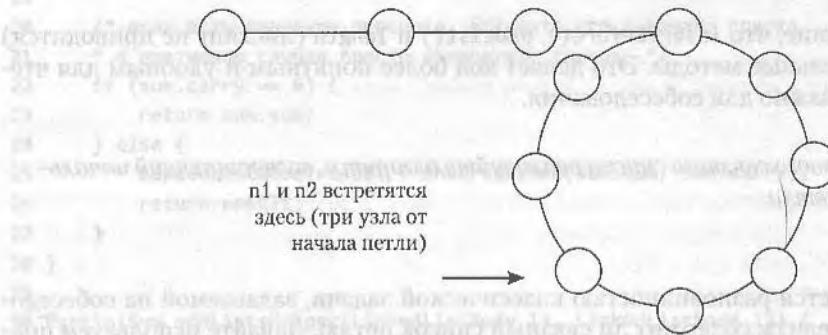
Мы знаем, что `FastRunner` перемещается в два раза быстрее, чем `SlowRunner`. Поэтому когда `SlowRunner` через k шагов попадет в петлю, `FastRunner` пройдет $2k$ шагов. Поскольку k существенно больше, чем длина петли, введем обозначение $K=\text{mod}(k, \text{LOOP_SIZE})$.

В каждом последующем шаге `FastRunner` и `SlowRunner` становятся на шаг (или два шага) ближе к цели. Поскольку система замкнута, когда А перемещается на q , оно становится на q шагов ближе к В.

Можно установить следующие факты:

1. `SlowRunner`: 0 шагов внутри петли.
2. `FastRunner`: K шагов.
3. `SlowRunner`: отстает от `FastRunner` на K шагов.
4. `FastRunner`: отстает от `SlowRunner` на `LOOP_SIZE-K` шагов.
5. `FastRunner` нагоняет `SlowRunner` со скоростью 1 шаг за единицу времени.

Когда же они встретятся? Если `FastRunner` на $LOOP_SIZE \cdot K$ шагов отстает от `SlowRunner`, а `FastRunner` нагоняет его со скоростью 1 шаг за единицу времени, они встретятся через $LOOP_SIZE \cdot K$ шагов. В этой точке они будут отстоять на K шагов от начала петли. Давайте назовем эту точку `CollisionSpot`.



Часть 3. Как найти начало петли?

Мы теперь знаем, что `CollisionSpot` — это K узел до начала петли. Поскольку $K = \text{mod}(k, LOOP_SIZE)$ (или $k = K + M * LOOP_SIZE$ для любого целого M), можно сказать, что до начала петли k узлов. Если узел $N - 2$ узла в петле из 5 элементов, то элементы 7, 12 и даже 397 принадлежат петле.

Поэтому 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`. Перемещение этих указателей заставит их столкнуться — на сей раз через k шагов — в точке `LoopStart`. Все, что нам нужно сделать, — возвратить 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 $LOOP_SIZE \cdot k$ узлов.

Если `FastPointer` двигается на 2 узла за одиничный шаг `SlowPointer`, указатели будут сближаться на каждом цикле и встретятся через $LOOP_SIZE \cdot k$ циклов. 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 один шаг.
3. Когда указатели встретятся, нужно передвинуть `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      /* Находим первую точку встречи LOOP_SIZE-k шагов
6       * по связному списку. */
7      while (fast != null && fast.next != null) {
8          slow = slow.next;
9          fast = fast.next.next;
10         if (slow == fast) { // Коллизия
11             break;
12         }
13     }
14
15     /* Ошибка - нет точки встречи, следовательно, нет петли */
16     if (fast == null || fast.next == null) {
17         return null;
18     }
19
20     /* Перемещаем медленный бегунок в начало списка (Head). Быстрый остается
   в точке встречи.
21     * Каждые k шагов от Loop Start. Если указатели продолжат
   движение с той же скоростью, то
22     * встретятся в точке Loop Start. */
23     slow = head;
24     while (slow != fast) {
25         slow = slow.next;
26         fast = fast.next;
27     }
28
29     /* Возвращаем точку начала петли. */
30     return fast;
31 }
```

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

Решение

Для этой задачи можно представить палиндром в виде списка $0 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 0$. Мы знаем, что если список является палиндромом, то начало и конец должны быть одинаковыми — это ключ к решению.

Решение 1. Инвертировать и сравнить

Первое решение — «перевернуть» связный список и сравнить с исходным. Если они одинаковы, значит, список является палиндромом.

Обратите внимание, что достаточно сравнить первые половины списков, и если они совпадают, то и вторые половины совпадут.

Решение 2. Итерационный подход

Необходимо найти связные списки, в которых первая половина списка является реверсом второй половины. Как это сделать? Путем инвертирования первой половины списка. Стек поможет нам в этом.

Поместим первую половину элементов в стек. Это можно делать разными способами, в зависимости от того, известен ли нам размер связного списка.

Если размер связного списка известен, можно пройти по первой половине элементов, помещая каждый элемент в стек. Обратите внимание — длина связного списка может быть нечетной.

Если мы не знаем размера связного списка, нужно использовать технику быстрого/медленного бегунка, описанную в начале главы. На каждом шаге мы помещаем данные медленного бегунка в стек. Когда быстрый бегунок дойдет до конца списка, медленный окажется в его середине. В этот момент в стеке находятся все элементы первой половины связного списка, но лежащие в обратном порядке.

Теперь нужно просто пройтись по оставшейся части списка. При каждой итерации мы будем сравнивать узел с вершиной стека. Если процесс закончился, не обнаружив разницы, значит, связный список — палиндром.

```

1  boolean isPalindrome(LinkedListNode head) {
2      LinkedListNode fast = head;
3      LinkedListNode slow = head;
4
5      Stack<Integer> stack = new Stack<Integer>();
6
7      /* Помещаем элементы первой половины связного списка в стек. Когда
8       * быстрый бегунок (который двигается со скоростью 2x) достигнет конца
9       * связного списка, мы узнаем, где середина */
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 число элементов, пропускаем средний элемент */
17     if (fast != null) {
18         slow = slow.next;
19     }
20
21     while (slow != null) {
22         int top = stack.pop().intValue();
23
24         /* Если значения различаются, это не палиндром */
25         if (top != slow.data) {
26             return false;
27         }
28         slow = slow.next;
29     }
30     return true;
31 }
```

Решение 3. Рекурсивный подход

Сначала небольшое замечание: в этом решении мы будем обозначать узлы kx , где k — значение узла, а x (может принимать значения f или b) указывает, к какому узлу мы обращаемся — из начала списка (f , *front*) или из конца (b , *back*). Например, узел $3b$ обращается к *back*-узлу со значением 3.

Как и другие задачи по связным спискам, эту задачу можно решить рекурсивно. Нужно сравнить элементы 0 и n , 1 и $n-1$, 2 и $n-2$ и так далее до центрального элемента. Например:

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

Чтобы применить данный подход, нужно сначала узнать, что мы дошли до 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 — в центре связного списка.

```
1 recurse(Node n, int length) {
2     if (length == 0 || length == 1) {
3         return [something]; // В середине
4     }
5     recurse(n.next, length - 2);
6     ...
7 }
```

Описанный метод лежит в основе метода *isPalindrome*. Суть алгоритма — сравнить узел i с узлом $n-i$ (проверка, является ли список палиндромом). Как это сделать?

Давайте представим, что стек вызовов выглядит так:

```
1 v1 = isPalindrome: список = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. длина = 7
2 v2 = isPalindrome: список = 1 ( 2 ( 3 ) 2 ) 1 ) 0. длина = 5
3 v3 = isPalindrome: список = 2 ( 3 ) 2 ) 1 ) 0. длина = 3
4 v4 = isPalindrome: список = 3 ) 2 ) 1 ) 0. длина = 1
5 возвращаем v3
6 возвращаем v2
7 возвращаем v1
8 возвращаем ?
```

В приведенном стеке вызовов каждый вызов проверяет, является ли список палиндромом, сравнивая его начальные узлы с соответствующими узлами 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 $3b$.

Если мы отмываем стек назад, то увидим, что:

- строка 4 «видит», что узел является средним (начиная с длины 1), и возвращает *head.next*. Значение *head* эквивалентно 3, поэтому *head.next* — узел $2b$;
- строка 3 сравнивает узел $2f$ с *returned_node* (значение предыдущего рекурсивного вызова), который является узлом $2b$. Если значения совпадают, мы возвращаем ссылку на узел $1b$ (*returned_node.next*) строке 2;

- строка 2 сравнивает узел `1f` с `returned_node` (узел `1b`). Если значения совпадают, передаем ссылку на `0b` (или `returned_node.next`) строке 1;
- строка 1 сравнивает узел `0f` с `returned_node` (узел `0b`). Если значения совпадают, возвращается `true`.

В общем виде: каждый вызов сравнивает начало с `returned_node` и затем передает в стек `returned_node.next`. Каждый узел i сравнивается с узлом $n-i$. Если в какой-либо точке они не совпадают, возвращается `false`.

Но подождите, как быть? Иногда нам нужно возвращать булево значение, иногда — узел?

Давайте создадим простой класс с двумя элементами — булевой переменной и узлом, — а возвращать будем экземпляр этого класса.

```
1 class Result {
2     public LinkedListNode node;
3     public boolean result;
4 }
```

Следующий пример иллюстрирует аргументы и возвращаемые значения из демонстративного списка:

```
1 isPalindrome: список = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. длина = 9
2 isPalindrome: список = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. длина = 7
3 isPalindrome: список = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. длина = 5
4 isPalindrome: список = 3 ( 4 ) 3 ) 2 ) 1 ) 0. длина = 3
5 isPalindrome: список = 4 ) 3 ) 2 ) 1 ) 0. длина = 1
6 возвращаем узел 3b, true
7 возвращаем узел 2b, true
8 возвращаем узел 1b, true
9 возвращаем узел 0b, true
10 возвращаем узел 0b, true
```

А реализация этого кода — дело техники:

```
1 Result isPalindromeRecurse(LinkedListNode head, int length) {
2     if (head == null || length == 0) {
3         return new Result(null, true);
4     } else if (length == 1) {
5         return new Result(head.next, true);
6     } else if (length == 2) {
7         return new Result(head.next.next,
8             head.data == head.next.data);
9     }
10    Result res = isPalindromeRecurse(head.next, length - 2);
11    if (!res.result || res.node == null) {
12        return res;
13    } else {
14        res.result = head.data == res.node.data;
15        res.node = res.node.next;
16    }
17 }
18 }
19
20 boolean isPalindrome(LinkedListNode head) {
```

```

21     Result p = isPalindromeRecurse(head, listSize(head));
22     return p.result;
23 }

```

Почему нам пришлось создавать специальный класс `Result`? Нет ли лучшего способа? Нет, по крайней мере, в Java.

Однако если вы реализуете код на C или C++, можно использовать двойной указатель:

```

1 bool isPalindromeRecurse(Node head, int length, Node** next) {
2     ...
3 }

```

Это немного уродливо, но работает.

Подход 2. Глобальное разбиение

Второй подход — глобальное разбиение пространства для связного списка. Каждый из ячеек в исходном массиве содержит ссылку на следующую ячейку. Поэтому мы можем разбить весь массив на несколько частей, каждая из которых будет иметь свой собственный головой. Например, если разбить список на две части, одна из которых будет состоять из ячеек с индексами от 0 до 4, а другая — от 5 до 9. Каждая из этих ячеек будет иметь ссылку на голову своей части.

Конечно, на собеседовании вас не заставят писать столь сложный и громоздкий глобально разбитый список, но это поможет вам лучше понять идею алгоритма.

```

1 // StackData — простой класс, который хранит набор данных о ячейке в списке
2 /* Класс не содержит элементы стека */
3 public class StackData {
4     public int start;
5     public int end;
6     public StackData(int start, int end) {
7         start = start;
8         end = end;
9         capacity = end - start + 1;
10    }
11    public StackData(int start, int end, StackData previous) {
12        start = start;
13        end = end;
14        previous = previous;
15        capacity = end - start + 1;
16    }
17    public StackData(int start, int end, StackData previous, StackData next) {
18        start = start;
19        end = end;
20        previous = previous;
21        next = next;
22        capacity = end - start + 1;
23    }
24 }
25
26 public boolean isPalindrome(ListNode head) {
27     if (head == null || head.next == null) {
28         return true;
29     }
30     StackData stackData = new StackData(0, 0);
31     int index = 0;
32     while (head != null) {
33         stackData = push(stackData, head);
34         head = head.next;
35         index++;
36     }
37     if (index < 2) {
38         return true;
39     }
40     StackData stackData2 = new StackData(index - 1, index);
41     int index2 = index - 1;
42     while (stackData2.start >= stackData.end) {
43         if (!isSame(stackData, stackData2)) {
44             return false;
45         }
46         stackData2 = pop(stackData2);
47         stackData = pop(stackData);
48         index2--;
49     }
50     return true;
51 }
52
53 private boolean isSame(StackData stackData, StackData stackData2) {
54     if (stackData.start >= stackData2.end) {
55         return true;
56     }
57     if (stackData.end <= stackData2.start) {
58         return true;
59     }
60     if (stackData.start > stackData2.end) {
61         return false;
62     }
63     if (stackData.end < stackData2.start) {
64         return false;
65     }
66     if (stackData.start >= stackData2.start && stackData.end <= stackData2.end) {
67         return true;
68     }
69     if (stackData.start <= stackData2.start && stackData.end >= stackData2.end) {
70         return true;
71     }
72     if (stackData.start <= stackData2.start && stackData.end > stackData2.end) {
73         return false;
74     }
75     if (stackData.start > stackData2.start && stackData.end <= stackData2.end) {
76         return false;
77     }
78     if (stackData.start > stackData2.start && stackData.end > stackData2.end) {
79         return false;
80     }
81     return true;
82 }
83
84 private StackData pop(StackData stackData) {
85     StackData result = stackData;
86     stackData = stackData.previous;
87     return result;
88 }
89
90 private StackData push(StackData stackData, ListNode head) {
91     StackData result = stackData;
92     stackData = new StackData(result.start, result.end + 1, stackData, head);
93     return result;
94 }

```

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 поддерживать эти стеки. Если нам нужно выделить определенное пространство для каждого стека, можно так и поступить. Но в этом случае один из стеков может исчерпать ресурсы, а другие будут практически пустыми.

Можно, конечно, использовать более гибкую систему разделения пространства, но это значительно усложняет задачу.

Подход 1: фиксированное разделение

Можно разделить массив на три равные части и разрешить стекам развитие в пределах ограниченного пространства. Обратите внимание, что далее мы будем описывать границы диапазонов с помощью скобок: квадратные скобки [] означают, что граничные значения входят в диапазон, а круглые скобки — значения не входят.

- Стек 1: $[0, n/3)$.
- Стек 2: $[n/3, 2n/3)$.
- Стек 3: $[2n/3, n)$.

Код для этого решения приведен ниже:

```

1 int stackSize = 100;
2 int[] buffer = new int [stackSize * 3];
3 int[] stackPointer = {0, 0, 0}; // указатели для отслеживания верхних элементов
4
5 void push(int stackNum, int value) throws Exception {
6     /* Проверяем, есть ли пространство */
7     if (stackPointer[stackNum] >= stackSize) {
8         throw new Exception("Недостаточно пространства.");
9     }
10    /* Находим индекс верхнего элемента массива + 1,
11       * и увеличиваем указатель стека */
12    int index = stackNum * stackSize + stackPointer[stackNum] + 1;
13    stackPointer[stackNum]++;
14    buffer[index] = value;
15 }
16
17 int pop(int stackNum) throws Exception {
18     if (stackPointer[stackNum] == 0) {
19         throw new Exception("Попытка использовать пустой стек");
20     }
21     int index = stackNum * stackSize + stackPointer[stackNum];
22     stackPointer[stackNum]--;

```

```

23     int value = buffer[index];
24     buffer[index] = 0;
25     return value;
26 }
27
28 int peek(int stackNum) {
29     int index = stackNum * stackSize + stackPointer[stackNum];
30     return buffer[index];
31 }
32
33 boolean isEmpty(int stackNum) {
34     return stackPointer[stackNum] == 0;
35 }

```

Если у нас есть дополнительная информация о назначении стеков, можно модифицировать алгоритм. 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 — гибкое выделение пространства для блоков стека. Когда один из стеков перестает помещаться в исходном пространстве, мы увеличиваем объем необходимого ресурса и при необходимости сдвигаем элементы.

Кроме того, можно создать массив таким образом, чтобы последний стек начинался в конце массива и заканчивался в начале, — «закольцевать» массив.

Впрочем, на собеседовании вас не заставят писать столь сложный код, поэтому мы ограничимся упрощенной версией (псевдокодом).

```

1 /* StackData - простой класс, который хранит набор данных о каждом стеке
2 * Класс не содержит элементы стека! */
3 public class StackData {
4     public int start;
5     public int pointer;
6     public int size = 0;
7     public int capacity;
8     public StackData(int _start, int _capacity) {
9         start = _start;
10        pointer = _start - 1;
11        capacity = _capacity;
12    }
13
14    public boolean isWithinStack(int index, int total_size) {
15        if (start + capacity <= total_size) { // нормальный размер
16            if (start <= index && index <= start + capacity) {
17                return true;
18            } else {
19                return false;
20            }
21        } else { // стек отсекается вокруг начала массива
22            int shifted_index = index + total_size;
23            if (start <= shifted_index &&
24                shifted_index <= start + capacity) {

```

продолжение ↗

```

25         return true;
26     } else {
27         return false;
28     }
29 }
30 }
31 }
32
33 public class QuestionB {
34     static int number_of_stacks = 3;
35     static int default_size = 4;
36     static int total_size = default_size * number_of_stacks;
37     static StackData [] stacks = {new StackData(0, default_size),
38         new StackData(default_size, default_size),
39         new StackData(default_size * 2, default_size)};
40     static int [] buffer = new int [total_size];
41
42 public static void main(String [] args) throws Exception {
43     push(0, 10);
44     push(1, 20);
45     push(2, 30);
46     int v = pop(0);
47     ...
48 }
49
50 public static int nextElement(int index) {
51     if (index + 1 == total_size) return 0;
52     else return index + 1;
53 }
54
55 public static int previousElement(int index) {
56     if (index == 0) return total_size - 1;
57     else return index - 1;
58 }
59
60 public static void shift(int stackNum) {
61     StackData stack = stacks[stackNum];
62     if (stack.size >= stack.capacity) {
63         int nextStack = (stackNum + 1) % number_of_stacks;
64         shift(nextStack); // выполняем сдвиг
65         stack.capacity++;
66     }
67     buffer[stack.start] = stack.data[stack.size - stack.capacity];
68     // Сдвигаем элементы в обратном порядке
69     for (int i = (stack.start + stack.capacity - 1) % total_size;
70         stack.isWithinStack(i, total_size);
71         i = previousElement(i)) {
72         buffer[i] = buffer[previousElement(i)];
73     }
74     buffer[stack.start] = 0;

```

```
76     stack.start = nextElement(stack.start); // перемещаем начало стека
77     stack.pointer = nextElement(stack.pointer); // перемещаем указатель
78     stack.capacity--; // устанавливаем оригинальный размер
79 }
80
81 /* Расширяем стек, сдвигаем остальные стеки */
82 public static void expand(int stackNum) {
83     shift((stackNum + 1) % number_of_stacks);
84     stacks[stackNum].capacity++;
85 }
86
87 public static void push(int stackNum, int value)
88 throws Exception {
89     StackData stack = stacks[stackNum];
90     /* Проверим, есть ли размер */
91     if (stack.size >= stack.capacity) {
92         if (numberOfElements() >= total_size) { // Totally full
93             throw new Exception("Недостаточно пространства.");
94         } else { // нужно выполнить сдвиг
95             expand(stackNum);
96         }
97     }
98     /* Находим индекс верхнего элемента в массиве + 1,
99      * и увеличиваем указатель стека */
100    stack.size++;
101    stack.pointer = nextElement(stack.pointer);
102    buffer[stack.pointer] = value;
103 }
104
105 public static int pop(int stackNum) throws Exception {
106     StackData stack = stacks[stackNum];
107     if (stack.size == 0) {
108         throw new Exception("Попытка использовать пустой стек");
109     }
110     int value = buffer[stack.pointer];
111     buffer[stack.pointer] = 0;
112     stack.pointer = previousElement(stack.pointer);
113     stack.size--;
114     return value;
115 }
116
117 public static int peek(int stackNum) {
118     StackData stack = stacks[stackNum];
119     return buffer[stack.pointer];
120 }
121
122 public static boolean isEmpty(int stackNum) {
123     StackData stack = stacks[stackNum];
124     return stack.size == 0;
125 }
126 }
```

В подобных задачах важно сосредоточиться на написании чистого и удобного в сопровождении кода. Вы должны использовать дополнительные классы, как мы сделали со `StackData`, а блоки кода нужно выделить в отдельные методы. Этот совет пригодится не только для прохождения собеседования, его можно использовать и в реальных задачах.

- 3.2. Как реализовать стек, в котором кроме стандартных функций `push` и `pop` будет использоваться функция `min`, возвращающая минимальный элемент?**

Оценка времени работы функций `push`, `pop` и `min` — $O(1)$.

Решение

Экстремумы изменяются не часто. Фактически, минимум может поменяться только при добавлении нового элемента.

Одно из решений — сравнивать добавляемые элементы с минимальным значением. Когда минимальное значение (`minValue`) удаляется из стека, приходится «перерывать» весь стек в поисках нового минимума. К сожалению, это нарушает ограничение на время выполнения — $O(1)$.

Давайте рассмотрим небольшой пример:

```
push(5); // стек - {5}, минимум - 5
push(6); // stack is {6, 5}, min is 5
push(3); // stack is {3, 6, 5}, min is 3
push(7); // stack is {7, 3, 6, 5}, min is 3
pop(); // pops 7. stack is {3, 6, 5}, min is 3
pop(); // pops 3. stack is {6, 5}. min is 5
```

Когда стек возвращается в предыдущее состояние ({6,5}), минимум также должен вернуться в предшествующее состояние (5). Это подталкивает нас ко второму варианту решения этой задачи.

Если мы будем отслеживать минимум в каждом состоянии, то легко узнаем минимальный элемент. Можно, например, записывать для каждого узла текущий минимальный элемент. Затем, чтобы найти `min`, достаточно «вытолкнуть» вершину и посмотреть, какой элемент является минимальным.

Как только элемент помещается в стек, локальное значение минимума становится глобальным.

```
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; // значение 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 один недостаток — если нужно обрабатывать огромный стек, то отслеживание минимального элемента потребует много ресурсов. Существует ли лучшее решение?

Оптимизировать код можно за счет использования дополнительного стека, который будет отслеживать 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? Предположим, что мы работаем с огромным стеком, первый вставленный элемент автоматически станет минимумом. В первом решении необходимо хранить n целых чисел, где n — размер стека. Во втором решении достаточно сохранить несколько фрагментов данных.

- 3.3.** Представьте стопку тарелок. Если стопка слишком высокая, она может развалиться. В реальной жизни, когда высота предыдущей стопки превысит некоторое значение, мы начали бы складывать тарелки в новую стопку. Реализуйте структуру данных `SetOfStacks`, имитирующую реальную ситуацию. Структура `SetOfStacks` должна состоять из нескольких стеков, новый стек создается, как только предыдущий достигнет порогового значения. Методы `SetOfStacks.push()` и `SetOfStacks.pop()` должны работать с общим стеком (со всей структурой) и должны возвращать те же значения, как если бы у нас был один большой стек.

Дополнительно:

Реализуйте функцию `popAt(int index)`, которая осуществляет операцию `pop` в указанный подстек.

Решение

В этой задаче 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 }
```

Мы знаем, что `push()` должен работать так же, как и с одиночным стеком, это означает, что `push()` нужно вызывать для последнего стека из массива стеков. При этом нужно действовать аккуратно: если последний стек не помещается в заданный объем, нужно создать новый стек. Код будет иметь примерно такой вид:

```
1 public void push(int v) {
2     Stack last = getLastStack();
3     if (last != null && !last.isFull()) {           // добавляем в последний стек
4         last.push(v);
5     } else {                                         // должны создать новый стек
6         Stack stack = new Stack(capacity);
7         stack.push(v);
8         stacks.add(stack);
9     }
10 }
```

Что же делает `pop()`? `pop()`, 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 public int pop() {
2     Stack last = getLastStack();
3     int v = last.pop();
4     if (last.size == 0) stacks.remove(stacks.size() - 1);
5     return v;
6 }
```

Дополнительно: реализация `popAt(int index)`

Данный метод реализовать немного сложнее. Если требуется вытолкнуть элемент из стека 1, то «низ» стека 2 нужно удалить и переместить его в стек 1. Затем аналогичную операцию нужно выполнить со стеками 3 и 2, 4 и 3 и т. д.

Данное решение не предназначено для работы со стеками, заполненными не полностью. Что делать с такими стеками? Не существует однозначного ответа, и вы должны обсудить 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. вышеупомянутый код */ }
14     public int pop() { /* см. вышеупомянутый код */ }
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) {

```

продолжение ↗

```

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 }

```

Данная задача не сложна, но требует написания довольно объемного кода. Интервьюер не потребует, чтобы вы написали весь код.

Хорошой стратегией будет разделение кода на методы. Например, в нашем коде существует метод `leftShift`, который вызывается в методе `popAt`. Такая уловка сделает код более прозрачным и понятным, вы сможете создать схему программы, а деталями заняться позже.

3.4. В задаче про Ханойскую башню задействованы 3 башни и N дисков разных размеров, которые нужно переместить (больший диск нельзя класть на меньший). Используя следующие ограничения:

- 1) за один раз можно переместить только один диск;
- 2) диски перемещаются только с вершины одной башни на другую башню;
- 3) диск можно положить только поверх большего диска.

Напишите программу перемещения дисков (с первой башни на последнюю), с использованием стеков.

Решение

Данная задача — хорошо подходит для метода «Базовый случай».

Давайте начнем со случая $n = 1$

1. Мы просто перемещаем диск 1 из Башни 1 на Башню 3.

Случай $n = 2$. Мы можем переместить диски 1 и 2 от Башни 1 до Башни 3? Да.

1. Переместить диск 1 с Башни 1 на Башню 2.
2. Переместить диск 2 с Башни 1 на Башню 3.
3. Переместить диск 1 с Башни 2 на Башню 3.

Заметьте, что мы используем Башню 2 как буфер, через который мы перемещаем другие диски на Башню 3.

Случай $n = 3$. Мы можем 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), теперь предположим, что это уже сделано, но в этом случае мы перемещали диски 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 $n = 4$. Мы можем переместить диски 1, 2, 3 и 4 с Башни 1 на Башню 3? Да!

1. Перемещаем диски 1, 2 и 3 на Башню 2. Как это сделать, было показано в примерах ранее.
2. Перемещаем диск 4 на Башню 3.
3. Перемещаем диски 1, 2 и 3 обратно на Башню 3.

Помните, что Башня 2 и Башня 3 эквивалентны. Перемещение дисков с Башни 2 на Башню 3 (которая служит буфером) аналогично перемещению дисков на Башню 2 с Башни 3, в этом случае Башня 3 служит буфером.

Данный подход сводится к рекурсивному алгоритму. Каждый раз мы выполняем действия, описанные следующим псевдокодом:

```

1 moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
2     /* Базовый случай */
3     if (n <= 0) return;
4
5     /* перемещаем верхние n - 1 дисков из Башни 1 на Башню 2, используя Башню 3
6     * как буфер. */
7     moveDisks (n - 1, Tower 1, Tower 2, Tower 3);
8
9     /* перемещаем вершину с Башни 1 на Башню 3
10    moveTop(Tower 1, Tower 3);
11
12    /* перемещаем верхние n - 1 дисков из Башни 2 на Башню 3, используя
13    * Башню 1 как буфер. */
14    moveDisks(n - 1, Tower 2, Tower 3, Tower 1);
15 }
```

Приведенный далее код предлагает более детальную реализацию того же алгоритма, но использует объектно-ориентированный подход:

```

1  public static 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 public 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("Ошибка перемещения диска " + 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         System.out.println("Перемещаем диск " + top + " из " + index() +
38             " на " + t.index());
39     }
40
41     public void moveDisks(int n, Tower destination, Tower buffer) {
42         if (n > 0) {
43             moveDisks(n - 1, buffer, destination);
44             moveTopTo(destination);
45             buffer.moveDisks(n - 1, destination, this);
46         }
47     }
48 }
```

Реализация башен в виде объектов не является необходимой, но делает код проще и понятнее.

3.5. Создайте класс `MyQueue`, который реализует очередь с использованием двух стеков.

Учитывая различия между очередью и стеком (FIFO против LIFO), нужно изменить методы `peek()` и `pop()` так, чтобы они работали в обратном порядке. Можно использовать второй стек, чтобы инвертировать порядок элементов (выталкиваем `s1` и помещаем все элементы в `s2`). В такой реализации каждая операция `peek()` или `pop()` приводит к выталкиванию всех элементов из `s1` в `s2`, после чего выполняется операция `peek/pop`, а затем все возвращаются обратно (с помощью `push`).

Этот алгоритм будет работать, но если мы выполняем операции `pop`/`peek` последовательно, элементы можно не перемещать. «Ленивый» способ — хранить элементы 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 удалить его из `stackOldest`. Если `stackOldest` пуст, то следует передать в этот стек все элементы из `stackNewest` в обратном порядке. Для вставки элемента нам нужно добавить его в `stackNewest`, поэтому новые элементы всегда будут 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        /* Push в stackNewest, в котором самые новые элементы
15         * будут на вершине */
16        stackNewest.push(value);
17    }
18
19    /* Перемещаем элементы из stackNewest в stackOldest. Это
20     * обычно и так сделано, поэтому мы можем выполнять операции над stackOldest.
21 */
22    private void shiftStacks() {
23        if (stackOldest.isEmpty()) {
24            while (!stackNewest.isEmpty()) {
```

подложение

```

24         stackOldest.push(stackNewest.pop());
25     }
26   }
27 }
28
29 public T peek() {
30     shiftStacks(); // Убедимся, что в stackOldest есть текущие элементы
31     return stackOldest.peek(); // получаем самый старый элемент
32 }
33
34 public T remove() {
35     shiftStacks(); // Убедимся, что в stackOldest есть текущие элементы
36     return stackOldest.pop(); // выталкиваем самый старый элемент
37 }
38 }
```

Во время собеседования вы можете обнаружить, что забыли некоторые API-вызовы. Не волнуйтесь, если это произошло. Большинство интервьюеров пойдут вам навстречу. Их в первую очередь интересует ваше видение задачи в целом.

- 3.6.** Напишите программу сортировки стека по возрастанию. Можно использовать дополнительные стеки для хранения элементов, но нельзя копировать элементы в другие структуры данных (например, в массив). Стек поддерживает следующие операции: push, pop, peek, isEmpty.

Решение

Один из подходов — реализовать алгоритм сортировки. Мы «перерываем» весь стек в поисках максимального элемента, выталкиваем его и помещаем в новый стек. Находим следующий максимальный элемент и выталкиваем его в новый стек. Такое решение потребует трех стеков: s_1 — исходный; s_2 — окончательный отсортированный стек; s_3 — буфер, используемый при поиске в s_1 . Каждый раз при поиске максимума мы должны вытолкнуть элемент из s_1 и поместить его в s_3 .

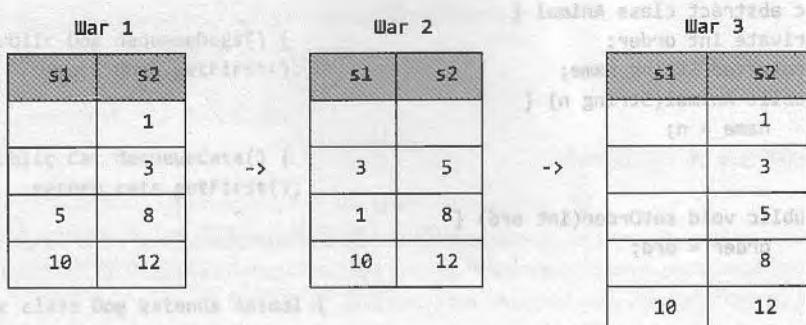
К сожалению, решение требует трех стеков. Можно как-то оптимизировать решение? Да!

Вместо многократного поиска минимума можно отсортировать s_1 , вставляя элементы в нужном порядке в s_2 . Как это реализовать?

Допустим, что у нас есть стеки:

s_1	s_2
	1
	3
5	8
10	12

Когда мы выталкиваем элемент 5 из s_1 , нам требуется вставить его в s_2 между элементами 3 и 8. Сначала 5 попадет из s_1 во временную переменную. Потом 3 и 1 переместятся из s_2 в s_1 , элемент 5 займет свое место в s_2 и 3 и 1 вернутся обратно.



Мы выталкиваем элемент t из $s1$ (в этом случае $t = 5$). Затем мы сравниваем t с вершиной $s2$, если $t < s2.pop()$, помещаем t в $s2$. В противном случае мы перемещаем элементы из $s2$ в $s1$, пока не найдем место для вставки t . Затем элементы копируются обратно.

```

1 public static Stack<Integer> sort(Stack<Integer> s) {
2     Stack<Integer> r = new Stack<Integer>();
3     while (!s.isEmpty()) {
4         int tmp = s.pop();
5         while (!r.isEmpty() && r.peek() > tmp) {
6             s.push(r.pop());
7         }
8         r.push(tmp);
9     }
10    return r;
11 }
```

Алгоритм требует $O(N^2)$ времени и $O(N)$ пространства.

- 3.7. 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, обслуживающую эту систему и реализующую операции enqueue, dequeueAny, dequeueDog и dequeueCat. Вы должны использовать встроенную структуру данных `LinkedList`.

Решение

Существует множество вариантов этой задачи. Например, можно работать с одной очередью. Тогда методы `dequeueAny`, `dequeueDog` и `dequeueCat` будут просматривать всю очередь, чтобы найти первую собаку или кошку. Трудоемкость такого решения велика, а эффективность мала.

Альтернативный подход прост и понятен — использовать разные очереди для собак и кошек, а поместить их в интерфейсный класс `AnimalQueue`. Кроме того, нужно хранить информацию о времени, чтобы знать, когда животное было поставлено в очередь. Когда мы вызываем `dequeueAny`, то проходим по очередям `dog` и `cat`, чтобы найти животное, которое дольше всех находится в очереди.

```

1 public abstract class Animal {
2     private int order;
3     protected String name;
4     public Animal(String n) {
5         name = n;
6     }
7
8     public void setOrder(int ord) {
9         order = ord;
10    }
11
12    public int getOrder() {
13        return order;
14    }
15
16    public boolean isOlderThan(Animal a) {
17        return this.order < a.getOrder();
18    }
19 }
20
21 public class AnimalQueue {
22     LinkedList<Dog> dogs = new LinkedList<Dog>();
23     LinkedList<Cat> cats = new LinkedList<Cat>();
24     private int order = 0; // временная привязка
25
26     public void enqueue(Animal a) {
27         /* Order используется для сортировки по времени, поэтому мы
28          * можем сравнить порядок вставки собаки и кошки */
29         a.setOrder(order);
30         order++;
31
32         if (a instanceof Dog) dogs.addLast((Dog) a);
33         else if (a instanceof Cat) cats.addLast((Cat)a);
34     }
35
36     public Animal dequeueAny() {
37         /* Смотрим на вершины очередей собак и кошек, выталкиваем стек
38          * с самым старым значением */
39         if (dogs.size() == 0) {
40             return cats.poll(); // Pop
41         } else if (cats.size() == 0) {
42             return dogs.poll(); // Pop
43         }
44         Dog dog = dogs.peek();
45         Cat cat = cats.peek();
46         if (dog.isOlderThan(cat)) {
47             return dogs.poll(); // Pop
48         } else {
49             return cats.poll(); // Pop
50         }
51     }

```

```

52     public Dog dequeueDogs() {
53         return dogs.getFirst();
54     }
55 }
56
57     public Cat dequeueCats() {
58         return cats.getFirst();
59     }
60 }
61
62 public class Dog extends Animal {
63     public Dog(String n) {
64         super(n);
65     }
66 }
67
68 public class Cat extends Animal {
69     public Cat(String n) {
70         super(n);
71     }
72 }

    public static boolean isEmpty(Deque<Animal> root) {
        if (root.getHeight(root) == 0) {
            return true;
        } else {
            /* Вызываем рекурсию */
            return Math.max(leftHeight, rightHeight) + 1;
        }
    }

    public static void main(String[] args) {
        Deque<Animal> root = new LinkedList<Animal>();
        Deque<Animal> left = new LinkedList<Animal>();
        Deque<Animal> right = new LinkedList<Animal>();

        left.addLast(new Dog("Buddy"));
        left.addLast(new Dog("Lucky"));
        left.addLast(new Cat("Fluffy"));

        right.addLast(new Cat("Whiskers"));
        right.addLast(new Cat("Patches"));

        root.addFirst(left);
        root.addFirst(right);

        System.out.println("Is empty? " + isEmpty(root));
    }
}

```

```

1 public abstract class Node<T> {
2     private int value;
3     protected Node left;
4     protected Node right;
5 }

```

4. Деревья и графы

- 4.1.** Реализуйте функцию, проверяющую сбалансированность двоичного дерева. Предположим, что дерево считается сбалансированным, если разница высот двух поддеревьев любого узла не превышает 1.

Решение

С этой задачей нам повезло — точно сформулировано, что имеется в виду под балансировкой дерева: для каждого узла два поддерева по высоте отличаются не более чем на один узел. Можно приступить к реализации решения, основанного на этом определении. Для этого достаточно рекурсивно пройтись по дереву и рассчитать высоту каждого узла.

```

1 public static int getHeight(TreeNode root) {
2     if (root == null) return 0; // Базовый случай
3     return Math.max(getHeight(root.left),
4         getHeight(root.right)) + 1;
5 }
6
7 public static boolean isBalanced(TreeNode root) {
8     if (root == null) return true; // Базовый случай
9
10    int heightDiff = getHeight(root.left) - getHeight(root.right);
11    if (Math.abs(heightDiff) > 1) {
12        return false;
13    } else { // Рекурсия
14        return isBalanced(root.left) && isBalanced(root.right);
15    }
16 }

```

Данное решение работает, но его нельзя назвать эффективным. Для каждого узла приходится рекурсивно исследовать все поддеревья. Это означает, что `getHeight` будет повторно вызываться на тех же самых узлах, а значит, оценка времени работы алгоритма составит $O(N^2)$.

Если внимательно посмотреть на `getHeight`, то станет ясно, что он может сразу проверить, сбалансировано ли дерево (одновременно с проверкой высот). Что делать, если обнаружится, что дерево не сбалансировано? Просто вернуть `-1`.

Улучшенный метод будет вызываться для вершины и проверять высоту каждого поддерева. Для каждого узла с помощью метода `checkHeight` мы рекурсивно получаем высоту левого и правого поддеревьев. Если поддерево сбалансировано, `checkHeight` возвращает высоту поддерева, когда оно сбалансировано, и `-1` в противоположном случае. При получении `-1` рекурсия прерывается.

Приведенный ниже код реализует данный алгоритм:

```

1 public static int checkHeight(TreeNode root) {
2     if (root == null) {
3         return 0; // Высота 0
4     }
5
6     /* Проверяем, сбалансировано ли левое поддерево. */
7     int leftHeight = checkHeight(root.left);
8     if (leftHeight == -1) {
9         return -1; // Не сбалансировано
10    }
11   /* Проверяем, сбалансировано ли правое поддерево. */
12   int rightHeight = checkHeight(root.right);
13   if (rightHeight == -1) {
14       return -1; // Не сбалансировано
15   }
16
17   /* Проверяем, сбалансирован ли текущий узел */
18   int heightDiff = leftHeight - rightHeight;
19   if (Math.abs(heightDiff) > 1) {
20       return -1; // Не сбалансирован
21   } else {
22       /* Возвращаем высоту */
23       return Math.max(leftHeight, rightHeight) + 1;
24   }
25 }
26
27 public static boolean isBalanced(TreeNode root) {
28     if (checkHeight(root) == -1) {
29         return false;
30     } else {
31         return true;
32     }
33 }
```

Этот код занимает $O(N)$ во времени и $O(\log N)$ в пространстве.

4.2. Разработайте алгоритм поиска маршрута между двумя узлами для направленного графа.

Решение

Данная задача может быть решена только простым обходом графа, например как поиск в глубину или в ширину. Поиск начинается с одного из узлов, и во время обхода выполняется проверка, найден ли другой узел. Мы должны отметить каждый узел как посещенный, чтобы не было циклов и повторных посещений узлов.

Приведенный ниже код выполняет итеративный обход графа в ширину:

```

1 public enum State {
2     Unvisited, Visited, Visiting;
3 }
```

продолжение ➔

```

4
5 public static boolean search(Graph g, Node start, Node end) {
6     // работает как стек
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(); // т.е., pop()
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 }
```

Можно заранее обсудить с интервьюером, какой метод обхода (поиск в глубину или в ширину) использовать для конкретной задачи. Поиск в глубину проще в реализации, поскольку основан на простой рекурсии. Поиск в ширину подходит для случаев, когда нужно найти кратчайший путь, поскольку поиск в глубину может зайти слишком глубоко по графу, прежде чем достигнет соседнего узла.

4.3. Напишите алгоритм создания бинарного дерева поиска с минимальной высотой для отсортированного (по возрастанию) массива.

Решение

Чтобы создать дерево минимальной высоты, 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.

Давайте построим дерево по этому правилу. Середина каждого подраздела массива становится корнем узла. Левая половина массива превращается в левое поддерево, а правая — в правое поддерево.

Можно использовать метод `root.insertNode(int v)`, который вставляет значение `v` с помощью рекурсивного процесса, начинающегося с корневого узла. В результате мы действительно получим дерево минимальной высоты, но такой алгоритм нель-

зя назвать эффективным. Каждая вставка потребует обхода дерева, в итоге оценка времени работы составит $O(N \log N)$.

Можно действовать по-другому — исключить дополнительные обходы, рекурсивно используя метод `createMinimalBST`. Данный метод получает только фрагмент массива и возвращает его корень минимального дерева.

Алгоритм следующий:

1. Вставьте средний элемент массива в дерево.
2. Вставьте левые элементы подмассива (в левое поддерево).
3. Вставьте правые элементы подмассива (в правое поддерево).
4. Повторите цикл.

Код, приведенный ниже, реализует этот алгоритм.

```

1 TreeNode createMinimalBST(int arr[], int start, int end) {
2     if (end < start) {
3         return null;
4     }
5     int mid = (start + end) / 2;
6     TreeNode n = new TreeNode(arr[mid]);
7     n.left = createMinimalBST(arr, start, mid - 1);
8     n.right = createMinimalBST(arr, mid + 1, end);
9     return n;
10 }
11
12 TreeNode createMinimalBST(int array[]) {
13     return createMinimalBST(array, 0, array.length - 1);
14 }
```

Хотя этот код не кажется сложным, можно легко допустить ошибку. Убедитесь, что вы тщательно протестировали все фрагменты кода.

- 4.4.** Для бинарного дерева поиска разработайте алгоритм, создающий связный список, состоящий из всех узлов заданной глубины (для дерева глубиной D должно получиться D связных списков).

Решение

Хотя на первый взгляд такая задача предполагает обход уровня за уровнем, но на самом деле так поступать не обязательно. Граф можно обходить любым удобным способом.

Давайте сделаем простую модификацию алгоритма обхода, передающую `level + 1` следующему вызову. Приведенный ниже код реализует обход с помощью поиска в глубину:

```

1 void createLevelLinkedList(TreeNode root,
2     ArrayList<LinkedList<TreeNode>> lists, int level) {
3     if (root == null) return; // базовый случай
4
5     LinkedList<TreeNode> list = null;
6     if (lists.size() == level) { // Уровня нет в списке
7         продолжение ↗
```

```

7     list = new LinkedList<TreeNode>();
8     /* Уровни всегда обходятся по порядку. Так, если мы впервые
9      * посещаем уровень i, мы уже просмотрели уровни
10     * от 0 до i - 1. Поэтому мы можем безопасно добавить уровень
11     * в конец.*/
12     lists.add(list);
13 } else {
14     list = lists.get(level);
15 }
16 list.add(root);
17 createLevelLinkedList(root.left, lists, level + 1);
18 createLevelLinkedList(root.right, lists, level + 1);
19 }
20
21 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
22 TreeNode root) {
23     ArrayList<LinkedList<TreeNode>> lists =
24         new ArrayList<LinkedList<TreeNode>>();
25     createLevelLinkedList(root, lists, 0);
26     return lists;
27 }

```

Существует альтернативное решение — модификация поиска в ширину. Начнем от корня, затем перейдем на уровень 2, потом на уровень 3 и т. д.

Таким образом, на уровень i мы попадем, только посетив все узлы на уровне $i - 1$. Чтобы понять, какие узлы относятся к уровню i , мы должны посмотреть на дочерние узлы уровня $i - 1$.

Приведенный ниже код реализует данный алгоритм:

```

1 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
2     TreeNode root) {
3     ArrayList<LinkedList<TreeNode>> result =
4         new ArrayList<LinkedList<TreeNode>>();
5     /* "Посещаем" корень */
6     LinkedList<TreeNode> current = new LinkedList<TreeNode>();
7     if (root != null) {
8         current.add(root);
9     }
10
11    while (current.size() > 0) {
12        result.add(current); // Добавляем предыдущий уровень
13        LinkedList<TreeNode> parents = current; // Переход на следующий уровень
14        current = new LinkedList<TreeNode>();
15        for (TreeNode parent : parents) {
16            /* Посещаем детей */
17            if (parent.left != null) {
18                current.add(parent.left);
19            }
20            if (parent.right != null) {
21                current.add(parent.right);
22            }

```

```

23     }
24 }
25 return result;
26 }
```

Какое из решений более эффективно? Оценка времени для обоих алгоритмов одинакова — $O(N)$. А как с памятью? Кажется, что с этой точки зрения второе решение более эффективно.

Первое решение использует $O(\log N)$ рекурсивных вызовов, и каждый вызов добавляет новый уровень стека. Второе, итеративное решение, не требует дополнительного пространства.

Оба решения возвращают $O(N)$ данных. Дополнительная память $O(\log N)$, расходуемая на рекурсивные вызовы, существенно меньше $O(N)$ — памяти, расходуемой на возврещение данных. Таким образом, оба решения одинаково эффективны, когда дело доходит до «О-большого».

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

Решение

Эту задачу можно решить двумя способами — симметричным обходом графа или использованием свойств 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.

Псевдокод этого метода будет иметь вид:

```

1 public static int index = 0;
2 public static 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 public static 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 }
```

Заметьте, что нам необходимо отследить, где окажется логический «конец» массива после того, как в него будут помещены все элементы графа.

Если проанализировать это решение, то легко прийти к выводу, что на самом деле массив не нужен. Мы используем его только для сравнения элемента с предыдущим. Но для этого достаточно отслеживать значение последнего элемента и сравнивать его с текущим.

Приведенный ниже код реализует модифицированную версию алгоритма:

```

1 public static int last_printed = Integer.MIN_VALUE;
2 public static boolean checkBST(TreeNode n) {
3     if (n == null) return true;
4
5     // Проверяем / рекурсия влево
6     if (!checkBST(n.left)) return false;
7
8     // Проверяем текущий
9     if (n.data < last_printed) return false;
10    last_printed = n.data;
11
12    // Проверяем / рекурсия вправо
13    if (!checkBST(n.right)) return false;
14
15    return true; // Все хорошо.
16 }
```

Если вы не любите использовать статические переменные, можно изменить код, добавив для целочисленного значения интерфейсный класс:

```

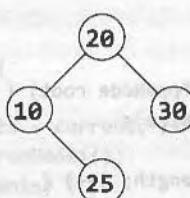
1 class WrapInt {
2     public int value;
3 }
```

Если вы пишете код на языке, который поддерживает передачу целых чисел по ссылке, воспользуйтесь этой возможностью.

Решение 2. Использование минимумов/максимумов

Второй вариант решения использует 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:

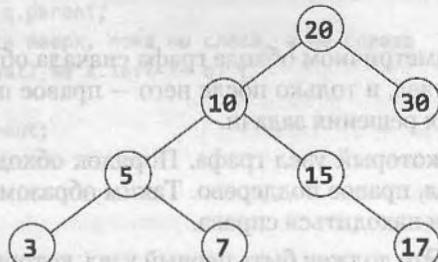


Хотя каждый узел этого дерева больше левого и меньше правого, это дерево нельзя назвать бинарным деревом поиска, поскольку 25 находится не на своем месте.

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.

Рассмотрим следующее дерево:



Начнем с диапазона ($\min = \text{INT_MIN}$, $\max = \text{INT_MAX}$), который обязательно содержит корень. Затем двигаемся влево, проверяя все узлы в интервале ($\min = \text{INT_MIN}$, $\max = 20$). Затем двигаемся в правую сторону и проверяем узлы в диапазоне ($\min = 10$, $\max = \text{INT_MAX}$).

Используя этот алгоритм, можно обойти все дерево. При движении влево обновляется максимум. При движении вправо обновляется минимум. Если проверка не пройдена, возвращается `false`.

Оценка времени для этого решения — $O(N)$, где N — количество узлов в дереве. Данное решение является лучшим, так как любой другой алгоритм должен будет проверять каждый узел дерева.

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 имеет вид:

```

1 boolean checkBST(TreeNode n) {
2     return checkBST(n, Integer.MIN_VALUE, Integer.MAX_VALUE);
3 }
4
5 boolean checkBST(TreeNode n, int min, int max) {
6     if (n == null) {
7         return true;
8     }
9     if (n.data <= min || n.data > max) {
10        return false;
11    }
12    if (!checkBST(n.left, min, n.data)) ||
13        !checkBST(n.right, n.data, max)) {
14        return false;
15    }
16 }
17 return true;
18 }
  
```

Не забывайте, что в рекурсивных алгоритмах нужно всегда проверять, что обработаны все варианты (например, `null`-случаи).

- 4.6. Напишите алгоритм поиска «следующего» узла для заданного узла в бинарном дереве поиска. Можно считать, 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. Эта информация понадобится вам для решения задачи.

Давайте рассмотрим 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? Это должен быть первый узел, который мы посетили бы, если бы делали обход этого поддерева. Таким образом, нам нужен «самый» левый узел на правом поддереве.

Но что делать, если у узла нет правого поддерева?

Если у узла n нет правого поддерева, нужно закончить обход поддерева n . Следует вернуться к предку n (назовем его, например, q).

Если n находится левее q , то q станет следующим узлом, который мы должны посетить (порядок обхода: левый \rightarrow текущий \rightarrow 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 полностью пройден? Если мы двигались слева направо, то левый узел должен быть полностью обследован, а его родитель — нет.

Псевдокод имеет примерно такой вид:

```

1 Node inorderSucc(Node n) {
2     if (n has a right subtree) {
3         return самый левый потомок правого поддерева
4     } else {
5         while (n - правый потомок n.parent) {
6             n = n.parent; // Идем вверх
7         }
8         return n.parent; // Предок еще не был пересечен
9     }
10 }
```

А что произойдет, если мы обойдем дерево полностью раньше, чем найдем левый дочерний элемент? Такая ситуация возможна, только если мы закончили обход. Если мы настолько ушли вправо, что не можем найти следующий элемент, следует возвратить `null`.

Код, представленный ниже, реализует этот алгоритм (и правильно обрабатывает `null`-случай):

```

1 public TreeNode inorderSucc(TreeNode n) {
2     if (n == null) return null;
3
4     /* Найден правый потомок -> возвращаем самый левый узел правого
5      * поддерева */
```

```

6     if (n.parent == null || n.right != null) {
7         return leftMostChild(n.right);
8     } else {
9         TreeNode q = n;
10        TreeNode x = q.parent;
11        // Поднимаемся вверх, пока мы слева, а не справа
12        while (x != null && x.left != q) {
13            q = x;
14            x = x.parent;
15        }
16        return x;
17    }
18 }
19
20 public TreeNode leftMostChild(TreeNode n) {
21     if (n == null) {
22         return null;
23     }
24     while (n.left != null) {
25         n = n.left;
26     }
27     return n;
28 }

```

Это не самая сложная задача, но написать превосходный код для ее решения достаточно трудно. В таких случаях полезно использовать псевдокод, чтобы схематически обрисовать в общих чертах все возможные случаи.

- 4.7.** Создайте алгоритм и напишите код поиска первого общего предка двух узлов бинарного дерева. Постарайтесь избежать хранения дополнительных узлов в структуре данных. Примечание: необязательно использовать бинарное дерево поиска.

Решение

Если бы речь шла о бинарном дереве поиска, то можно было бы модифицировать оператор `find` для работы с двумя узлами, чтобы отследить направление движения. К сожалению, в задаче есть ограничение на использование бинарного дерева поиска, поэтому нужно искать другие подходы.

Предположим, что мы хотим найти общего предка для узлов `r` и `q`. Остается решить один вопрос — имеют ли узлы нашего дерева связь с предками.

Решение 1. С учетом связи с предками

Если у каждого узла есть связь с предками, можно отследить пути `r` и `q`, вплоть до их пересечения. Однако такой подход может нарушить некоторые исходные предположения, поскольку потребует обеспечить: а) возможность помечать узлы как посещенные (`isVisited`) или б) возможность хранить некоторые данные в дополнительной структуре, например в хэш-таблице.

Решение 2. Без учета связи с предками

Можно проследить цепочки, в которых p и q находятся на одной стороне. Если p и q находятся слева от узла, то, чтобы найти общего предка, следует двигаться влево. Если они находятся справа, то и двигаться нужно вправо. Как только p и q окажутся по разные стороны от узла — ближайший общий предок найден.

Код, приведенный ниже, реализует этот подход:

```

1 /* Возвращает true, если p - потомок корня */
2 boolean covers(TreeNode root, TreeNode p) {
3     if (root == null) return false;
4     if (root == p) return true;
5     return covers(root.left, p) || covers(root.right, p);
6 }
7
8 TreeNode commonAncestorHelper(TreeNode root, TreeNode p,
9 TreeNode q) {
10    boolean is_p_on_left = covers(root.left, p);
11    boolean is_q_on_left = covers(root.left, q);
12
13    /* Если p и q находятся на разных сторонах, возвращаем корень */
14    if (is_p_on_left != is_q_on_left) return root;
15
16    /* Если они находятся на одной стороне, "путешествуем" по ней */
17    TreeNode child_side = is_p_on_left ? root.left : root.right;
18    return commonAncestorHelper(child_side, p, q);
19 }
20
21 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
22     if (!covers(root, p) || !covers(root, q)) { // Error check
23         return null;
24     }
25     return commonAncestorHelper(root, p, q);
26 }
```

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 добиться уже невозможно — мы должны проанализировать каждый узел в дереве. Однако мы можем уменьшить время за счет постоянного множителя.

Решение 3. Оптимизация

Хотя решение 2 является оптимальным, его можно еще чуть-чуть улучшить. `Covers` ищет все узлы под `root` для p или q , включая узлы в каждом поддереве (`root.left` и `root.right`). Затем `covers` выбирает одно из поддеревьев и ищет все его узлы. То есть каждое поддерево проходится много раз.

Как сделать так, чтобы дерево при поиске p и q просматривалось только один раз?

Можно выталкивать ранее найденные узлы в стек. Общая логика решения остается такой же, как в предыдущем случае.

Мы рекурсивно проходимся по всему дереву с помощью функции `commonAncestor(TreeNode root, TreeNode p, TreeNode q)`, которая возвращает следующие значения:

- `p`, если поддерево корня содержит `p` (и не содержит `q`);
- `q`, если поддерево корня содержит `q` (и не содержит `p`);
- `null`, если ни `p`, ни `q` нет в поддереве;
- в противном случае возвращаем предка `p` и `q`.

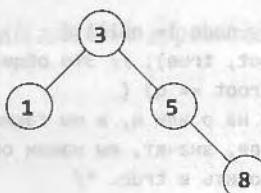
Нахождение общего предка `p` и `q` — финальная стадия алгоритма. Когда и `commonAncestor(n.left, p, q)` и `commonAncestor(n.right, p, q)` возвращают не-`null` значения, это означает, что `n` является общим предком для `p` и `q`.

Следующий код содержит ошибку. Вы сможете ее найти?

```

1 /* Ниже код с ошибкой!!! */
2 TreeNode commonAncestorBad(TreeNode root, TreeNode p, TreeNode q) {
3     if (root == null) {
4         return null;
5     }
6     if (root == p && root == q) {
7         return root;
8     }
9
10    TreeNode x = commonAncestorBad(root.left, p, q);
11    if (x != null && x != p && x != q) { // Найден потомок
12        return x;
13    }
14
15    TreeNode y = commonAncestorBad(root.right, p, q);
16    if (y != null && y != p && y != q) { // Найден предок
17        return y;
18    }
19
20    if (x != null && y != null) { // p и q найдены в разных поддеревьях
21        return root;           // Это общий предок
22    } else if (root == p || root == q) {
23        return root;
24    } else {
25        /* Если или x или y - не null, возвращаем не-null значение */
26        return x == null ? y : x;
27    }
28 }
```

Программа выдает ошибку, если в дереве нет узла. Например, взгляните на следующее дерево:



Предположим, что мы сделали вызов функции `commonAncestor(node 3, node 5, node 7)`. Узла 7 не существует, что приводит к ошибке:

```

1 commonAncestor(node 3, node 5, node 7)           // --> 5
2     calls commonAncestor(node 1, node 5, node 7)   // --> null
3     calls commonAncestor(node 5, node 5, node 7)   // --> 5
4         calls commonAncestor(node 8, node 5, node 7) // --> null

```

Другими словами, когда мы вызываем `commonAncestor` на правом поддереве, код вернет узел 5 (и это правильно). Проблема заключается в следующем: при обнаружении общего предка `p` и `q` функция не может различить:

- 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 правильный ответ должен быть `null`.

Нам нужно разделить эти два случая, а представленный ниже код решает эту проблему, возвращая два значения — узел и флаг, по которому можно понять, является ли узел общим предком.

```

1 public static class Result {
2     public TreeNode node;
3     public boolean isAncestor;
4     public Result(TreeNode n, boolean isAnc) {
5         node = n;
6         isAncestor = isAnc;
7     }
8 }
9
10 Result commonAncestorHelper(TreeNode root, TreeNode p, TreeNode q){
11     if (root == null) {
12         return new Result(null, false);
13     }
14     if (root == p && root == q) {
15         return new Result(root, true);
16     }
17
18     Result rx = commonAncestorHelper(root.left, p, q);
19     if (rx.isAncestor) { // Найден общий предок
20         return rx;
21     }
22
23     Result ry = commonAncestorHelper(root.right, p, q);
24     if (ry.isAncestor) { // Найден общий предок
25         return ry;
26     }
27
28     if (rx.node != null && ry.node != null) {
29         return new Result(root, true); // Это общий предок
30     } else if (root == p || root == q) {
31         /* Если мы находимся на p или q, и мы также нашли
32          * эти узлы в поддереве, значит, мы нашли общего предка
33          * и флаг нужно установить в true. */

```

```

34     boolean isAncestor = rx.node != null || ry.node != null ?
35         true : false;
36     return new Result(root, isAncestor);
37 } else {
38     return new Result(rx.node!=null ? rx.node : ry.node, false);
39 }
40 }
41
42 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
43     Result r = commonAncestorHelper(root, p, q);
44     if (r.isAncestor) {
45         return r.node;
46     }
47     return null;
48 }

```

Поскольку проблема возникает, только когда p или q нет в дереве, простое альтернативное решение — просмотреть все дерево, дабы удостовериться, что существуют оба узла.

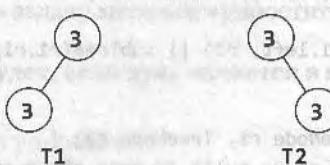
- 4.8.** Дано: два очень больших бинарных дерева: T_1 — с миллионами узлов и T_2 — с сотнями узлов. Создайте алгоритм, проверяющий, является ли T_2 поддеревом T_1 . Дерево T_2 считается поддеревом T_1 , если существует такой узел n в T_1 , что поддерево, «растущее» из n , идентично дереву T_2 . (Если вы вырежете дерево в узле n и сравните его с T_2 , они окажутся идентичны.)

Решение

Подобные задачи нужно решать сначала для небольшого количества данных. Это позволит сформировать общую концепцию решения.

В упрощенной задаче можно создать строку, эквивалентную симметричному (in-order) и прямому (pre-order) порядкам обхода дерева. При прямом обходе T_2 будет являться подстрокой прямого обхода дерева T_1 , при симметричном обходе T_2 — это подстрока симметричного обхода T_1 . Время, затрачиваемое на поиск подстроки, растет линейно, поэтому данный алгоритм довольно эффективен.

Не забудьте добавить в строки специальный знак, сигнализирующий, что левый или правый узел принял значение `null`. Иначе возникнет проблема с идентификацией случаев.



Симметричный и прямой обходы для этих деревьев совпадают:

T_1 , in-order: 3, 3

T_1 , pre-order: 3, 3

T_2 , in-order: 3, 3

T_2 , pre-order: 3, 3

Стоит нам пометить NULL-значения, и два дерева станут различаться:

```
T1, in-order: 0, 3, 0, 3, 0
T1, pre-order: 3, 3, 0, 0, 0
T2, in-order: 0, 3, 0, 3, 0
T2, pre-order: 3, 0, 3, 0, 0
```

Данное решение подходит для простого случая, но в нашей задаче используются большие объемы данных. Создание копий деревьев потребует огромного объема памяти.

Альтернативный подход

Альтернативный подход — поиск по большему дереву t_1 . Если узел t_1 совпадает с корнем t_2 , то нужно вызвать метод `treeMatch`, который сравнивает два поддерева, чтобы проверить, являются ли они идентичными.

Анализ времени выполнения достаточно трудоемок. Наивно думать, что оценка времени составит $O(nm)$, где n — количество узлов в t_1 , а m — количество узлов в t_2 . Арифметически это кажется правдоподобным, но на практике дело обстоит не совсем так.

Мы не вызываем метод `treeMatch` для каждого узла t_2 . Мы вызываем его k раз, где k показывает, сколько раз корень t_2 встречается в t_1 . Поэтому $O(n + km)$ — более правильная оценка.

На самом деле эта оценка несколько завышена. Даже если корень совпал, мы выходим из `treeMatch`, как только обнаруживается различие между t_1 и t_2 . Поэтому фактически 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) { // Пустое дерево - всегда поддерево
3         return true;
4     }
5     return subTree(t1, t2);
6 }
7
8 boolean subTree(TreeNode r1, TreeNode r2) {
9     if (r1 == null) {
10         return false; // большое дерево пустое & поддерево не найдено
11     }
12     if (r1.data == r2.data) {
13         if (matchTree(r1,r2)) return true;
14     }
15     return (subTree(r1.left, r2) || subTree(r1.right, r2));
16 }
17
18 boolean matchTree(TreeNode r1, TreeNode r2) {
19     if (r2 == null && r1 == null) // если оба пусты
20         return true; // ничего нет в поддереве
21
22     // если одно, но не оба, пусто
23     if (r1 == null || r2 == null) {
24         return false;
25     }
```

```

26
27     if (r1.data != r2.data)
28         return false; // данные не совпадают
29     return (matchTree(r1.left, r2.left) &&
30             matchTree(r1.right, r2.right));
31 }
32 }
```

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 требования к памяти очень важны, когда речь заходит о масштабируемости.
2. Простое решение потребует $O(n + m)$ времени, а альтернативное — $O(nm)$. Хотя вероятность столь плохого результата невелика, но нам нужно ее учитывать.
3. Немного сложнее с временем выполнения $O(n + km)$, как мы уже выяснили ранее (k — коэффициент, показывающий, сколько раз корень T_2 встречается в T_1). Предположим, что данные в деревьях T_1 и T_2 — случайные величины в диапазоне от 0 до p . Тогда k стремится к n/p . Почему? Каждый из n узлов в T_1 может быть равен корню $T_2.root$ с вероятностью $1/p$. Пусть $p = 1000$, $n = 1000000$ и $m = 100$. Тогда нам понадобится выполнить около 1 100 000 проверок узлов ($1000000 = 1000000 + 100 * 1000000 / 1000$).
4. Более сложные математические расчеты позволят дополнительно уточнить это значение. Ранее мы предположили, что при вызове `treeMatch` нам придется обойти m узлов T_2 , 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 времени выполнения. Однако стоит дополнительно поговорить на эту тему с интервьюером.

- 4.9.** Дано бинарное дерево, в котором каждый узел содержит число. Разработайте алгоритм, выводящий на печать все пути, сумма значений которых соответствует заданной величине. Обратите внимание, что путь может начинаться и заканчиваться в любом месте дерева.

Решение

Допустим, мы будем решать задачу методом «упростить и обобщить».

Часть 1: упрощение. Что будет, если путь начнется в корне, а закончится где угодно?

В этом случае задача сильно упрощается.

Можно начать с корня и «пойти» влево или вправо, вычисляя сумму для каждого пути. Когда мы найдем сумму, то выведем значение для текущего пути. Обратите внимание, что обход нельзя останавливать после получения суммы, удовлетворяющей нашему условию. Почему? Потому что на пути могут встречаться как +1, так и -1 (или любая другая последовательность узлов, вклад которых в сумму может оказаться равным 0), а сумма полного пути все равно окажется равной заданной величине `sum`.

Например, если `sum = 5`, у нас будут следующие пути:

- `p = {2, 3}`
- `q = {2, 3, -4, -2, 6}`

Если мы остановимся, как только получим $2 + 3$, то не найдем оставшуюся часть пути. А нам необходимо вывести каждый возможный путь.

Часть 2: обобщение. Путь может начинаться где угодно

Предположим, что путь может начинаться где угодно. В этом случае проведем небольшую модификацию. Для каждого узла теперь придется проверять, не найдена ли сумма. Таким образом, вместо выяснения, «начинает ли этот узел путь с нужной суммой», необходимо выяснить, «заканчивает ли этот узел путь с нужной суммой».

Когда мы проходим по каждому узлу `n`, то передаем функции полный путь от корня до `n`. Эта функция суммирует все узлы в обратном порядке — от `n` до `root` (корень).

Когда сумма подпути равна `sum`, мы выводим это значение:

```
1 public void findSum(TreeNode node, int sum, int[] path, int level) {  
2     if (node == null) {  
3         return;  
4     }  
5     /* Вставляем текущий узел в путь */  
6     path[level] = node.data;  
7  
8     /* Ищем путь с суммой, которая заканчивается на этом узле */  
9     int t = 0;  
10    for (int i = level; i >= 0; i--) {  
11        t += path[i];  
12        if (t == sum) {  
13            print(path, i, level);  
14        }  
15    }  
16  
17    /* Ищем узлы ниже этого */  
18    findSum(node.left, sum, path, level + 1);  
19    findSum(node.right, sum, path, level + 1);  
20  
21    /* Удаляем текущий узел из пути. Это не является необходимым,  
22     * мы можем просто игнорировать это значение */  
23    path[level] = Integer.MIN_VALUE;  
24}  
25 }  
  
26  
27 public void findSum(TreeNode node, int sum) {  
28     int depth = depth(node);  
29     int[] path = new int[depth];  
30     findSum(node, sum, path, 0);  
31 }  
32  
33 public static void print(int[] path, int start, int end) {  
34     for (int i = start; i <= end; i++) {  
35 }
```

```

35     System.out.print(path[i] + " ");
36   }
37   System.out.println();
38 }
39
40 public int depth(TreeNode node) {
41   if (node == null) {
42     return 0;
43   } else {
44     return 1 + Math.max(depth(node.left), depth(node.right));
45   }
46 }

```

Как оценить время выполнения этого алгоритма? Можно предположить, что время выполнения составит примерно $O(n \log(n))$, поскольку для n узлов потребуют $\log(n)$ действий на каждом шаге.

Если вы сомневаетесь в правильности этой оценки, давайте поговорим на языке математики. Заметьте, что у нас есть 2^r узлов на уровне r .

$$\begin{aligned}
 & 1 * 2^1 + 2 * 2^2 + 3 * 2^3 + 4 * 2^4 + \dots + d * 2^d = \\
 & = \text{sum}(r * 2^r, r \text{ from } 0 \text{ to } \text{depth}) \\
 & = 2 * (d - 1) * 2^d + 2 \\
 n & = 2^d \\
 d & = \log(n)
 \end{aligned}$$

Если вспомнить, что $2^{\log(x)} = x$, то выражение можно упростить:

$$\begin{aligned}
 & O(2 * (\log(n) - 1) * 2^{\log(n)} + 2) = \\
 & = O(2 * (\log n - 1) * n) \\
 & = O(n \log(n))
 \end{aligned}$$

Аналогичным образом можно получить оценку требуемого пространства — $O(n \log(n))$.

5. Поразрядная обработка

- 5.1.** Дано: два 32-битных 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 . Можно считать, что j и i имеют такие значения, что число M обязательно поместится в этот промежуток. Если $M = 10011$, для его размещения понадобится 5 битов между j и i . Если $j = 3$ и $i = 2$, число M не поместится в указанный промежуток.

Пример:

Ввод: $N = 10000000000, M = 10011, i = 2, j = 6$

Вывод: $N = 10001001100$

Решение

Эту задачу можно решить за три шага:

1. Очистите в N биты с j по i .
2. Сдвиньте M так, чтобы оно оказалось под позициями j до i .
3. Соедините M и N .

Самая сложная часть — шаг 1. Как очистить биты в N ? Это можно сделать с помощью маски. Маска должна содержать единицы во всех позициях, кроме интервала с j по i (в этих позициях должны стоять нули). Давайте сначала создадим левую половину маски, затем — правую.

```

1 int updateBits(int n, int m, int i, int j) {
2     /* Создаем маску очистки битов от i до j в п
3     /* ПРИМЕР: i = 2, j = 4. Результат должен быть 11100011.
4     * Для простоты будем считать, что у нас всего 8 бит.
5     */
6     int allOnes = ~0; // эквивалентно последовательности единиц
7
8     // единицы до j, потом 0s. Левая часть = 11100000
9     int left = allOnes << (j + 1);
10    // Удаляем текущий участок пути. Для него нужна маска.
11    // единицы после i. Правая часть = 00000011
12    int right = ((1 << i) - 1);
13
14    // Все единицы, за исключением нулей между i и j. маска = 11100011
15    int mask = left | right;
16
17    /* очищаем биты с j по i, затем помещаем m сюда */
18    int n_cleared = n & mask; // очищаем биты
19    int m_shifted = m << i; // помещаем m на подготовленную позицию
20
21    return n_cleared | m_shifted; // операция OR, все готово!
22 }
```

В задачах, работающих с битами, очень легко допустить ошибку, поэтому всегда тщательно проверяйте код.

- 5.2. Дано: вещественное число в интервале между 0 и 1 (например, 0,72), которое было передано как double, запишите его двоичное представление. Если для 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 0.101₂ можно записать как

$$0.101_2 = 1 * (1/2^1) + 0 * (1/2^2) + 1 * (1/2^3)$$

Чтобы вывести десятичную часть, мы можем умножить число на 2 и сравнить $2n$ с 1. Это, по существу, сдвигает дробную сумму:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * (1/2^0) + 0 * (1/2^1) + 1 * (1/2^2) \\ &= 1.01 \end{aligned}$$

Если $r \geq 1$, то сразу понятно, что после десятичной точки в n находится 1. Таким же образом можно проверить каждую цифру.

```

1 public static 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         /* Устанавливаем лимит строки: 32 символа */
10        if (binary.length() > 32) {
11            return "ERROR";
12        }
13        double r = num * 2;
14        if (r >= 1) {
15            binary.append(1);
16            num = r - 1;
17        } else {
18            binary.append(0);
19            num = r;
20        }
21    }
22    return binary.toString();
23 }
24 }
```

Другой подход — вместо умножения числа на 2 и сравнения с 1 делать сравнение числа с 0.5, потом с 0.25 и т. д. Приведенный ниже код демонстрирует реализацию этого подхода:

```

1 public static String printBinary2(double num) {
2     if (num >= 1 || num <= 0) {
3         return "ERROR";
4     }
5
6     StringBuilder binary = new StringBuilder();
7     double frac = 0.5;
8     binary.append(".");
9     while (num > 0) {
10         /* Лимит строки: 32 символа */
11         if (binary.length() > 32) {
12             return "ERROR";
13         }
14         if (num >= frac) {
15             binary.append(1);
16             num -= frac;
17         } else {
18             binary.append(0);
19         }
20         frac /= 2;
21     }
22     return binary.toString();
23 }
```

Оба варианта одинаково хороши, используйте тот, который вам больше нравится. Что бы вы ни выбрали, убедитесь, что предусмотрели все возможные случаи, и обсудите их с интервьюером.

5.3. Дано: положительное число. 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, поразрядная обработка, арифметический подход и т. д. Обратите внимание, что арифметический подход основан на битовой обработке. Поэтому сначала стоит заняться манипуляцией битами, а только затем перейти к арифметическому методу.

Метод грубой силы

Самый простой способ — метод 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!

Давайте начнем с кода для `getNext` (возвращает следующее число), а затем перейдем к `getPrev` (возвращает предыдущее число).

Поразрядная обработка для getNext

Подумаем, каким должно быть следующее число? Пусть дано число 13948, его двоичное представление имеет вид:

1	1	0	1	1	0	0	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	0

Нам нужно получить ближайшее большее число и сохранить исходное.

Примечание: дано некоторое число n с позициями битов i и j , и мы инвертируем значение бита i из 1 в 0, а бита j — из 0 в 1. Если $i > j$, значит, число n уменьшилось. Если $i < j$, значит, число n увеличилось.

Из этого можно вывести следующие факты:

Если мы инвертируем значение одного из битов из 0 в 1, то необходимо инвертировать другой бит из 1 в 0.

Число окажется больше, если бит, преобразованный из 0 в 1, находился левее бита, преобразованного из 1 в 0.

Результирующее число должно быть больше, но ненамного. То есть нам необходимо инвертировать самый правый 0, у которого справа есть 1.

Другой способ — инвертировать самые правые неконечные нули. В нашем примере самый правый неконечный нуль — это бит 7. Назовем эту позицию p .

Шаг 1: инвертируем правые «не завершающие» нули.

1	1	0	1	1	0	1	1	1	1	1	1	1	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	0

Давайте рассмотрим работу этого алгоритма подробнее:

Шаг 2: очищаем биты, находящиеся справа от p ($c0 = 2$, $c1 = 5$, $p = 7$).

1	1	0	1	1	0	1	0	0	0	0	0	0	0	0
13	12	11	10	9	8	7	6	5	4	3	2	1	0	0

Для очистки данных битов нужно создать маску, представляющую собой последовательность единиц, после которой стоит p нулей:

$a = 1 \ll pos$; // все нули за исключением 1 на позиции p .

$b = a - 1$; // все нули после p единиц.

$mask = \sim b$; // все единицы после p нулей.

$n = n \& mask$; // очищаем p самых правых битов.

В сокращенной записи:

$n \&= \sim((1 \ll pos) - 1)$.

Шаг 3: устанавливаем младшие $c1 - 1$ бит в единицу.

1	1	0	1	1	0	1	0	0	0	1	1	1	1	1
13	12	11	10	9	8	7	6	5	4	3	2	1	0	0

Чтобы вставить справа $c1 - 1$ единиц, выполните следующие действия:

```
a = 1 << (c1 - 1); // 0s with a 1 at position c1
b = a - 1;          // 0s with 1s at positions 0 through c1 - 1
n = n | b;          // inserts 1s at positions 0 through 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`:

```
1 public int getNext(int n) {
2     /* Вычисляем 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    while ((c & 1) == 1) {
11        c1++;
12        c >>= 1;
13    }
14 }
15
16 /* Ошибка: если n == 11..1100...00, значит, тут нет большего числа
17 * с таким же числом единиц */
18 if (c0 + c1 == 31 || c0 + c1 == 0) {
19     return -1;
20 }
21
22 int p = c0 + c1;           // позиция для самых правых ненеончных нулей
23
24 n |= (1 << p);           // Зеркально отражаем самые правые ненеончные 0
25 n &= ~((1 << p) - 1);    // Очищаем все биты справа от p
26 n |= (1 << (c1 - 1)) - 1; // Вставляем (c1-1) единиц справа.
27
28 }
```

Поразрядная обработка для `getPrev`

Для реализации `getPrev` можно использовать аналогичный подход:

1. Вычислим $c0$ и $c1$. $c1$ — это число завершающих нулей, а $c0$ — размер блока нулей, расположенных левее завершающих нулей.
2. Замените все незавершающие единицы на нули. Это будут позиции $p = c1 + c0$.
3. Очистите все биты, находящиеся справа от бита p .
4. Установите 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. Можно объединить оба этих шага.

Давайте 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 & 3: очищаем биты от 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 0 - p.
```

Шаг 4: вставляем $(c1 + 1)$ 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 на позиции (c1 + 1)
int b = a - 1; // нули в сопровождении (c1 + 1) единиц
int c = b << (c0 - 1); // (c1+1) единиц в сопровождении c0-1 нулей.
n |= c;
```

Далее приведен код, реализующий `getPrev`:

```
1 int getPrev(int n) {
2     int temp = n;
3     int c0 = 0;
4     int c1 = 0;
5     while (((temp & 1) == 1) && (temp != 0)) {
6         c1++;
7         temp >>= 1;
8     }
9     if (temp == 0) return -1;
10    while ((temp & 1) == 0 && (temp != 0)) {
11        c0++;
12        temp >>= 1;
13    }
14    int p = c0 + c1; // позиция "самой правой" незавершающей единицы
15    n &= ((~0) << (p + 1)); // очищаем бит p
16
17
18
19
```

продолжение

```

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 конечных нулей, $c1$ — размер следующего блока единиц и $p = c0 + c1$, то наше решение можно сформулировать так:

1. Устанавливаем p -й бит в 1.
2. Устанавливаем все биты после p в 0.
3. Устанавливаем биты с 0 до $c1 - 1$ в 1.

Быстрый способ выполнить шаги 1 и 2 — установить завершающие нули в 1, а затем прибавить 1. Сложение с единицей инвертирует завершающие единицы, таким образом, мы получим 1 в бите p , за которым будут идти p нулей. Можно решить эту задачу арифметически.

```

n += 2c0 - 1;      // Установить завершающие 0 в 1
n += 1;           // Отразить первые p единиц в 0, и поместить 1 в бит p
```

Теперь, чтобы осуществить шаг 3 арифметически, нужно:

```
n += 2c1 - 1 - 1; // Установить c1 - 1 нулей в 1.
```

Упростим арифметические выражения:

```

next = n + (2c0 - 1) + 1 + (2c1 - 1 - 1) =
      = n + 2c0 + 2c1 - 1 - 1
```

В итоге получаем простой код:

```

1 int getNextArith(int n) {
2     /* ... то же вычисление c0 c1 как и раньше ... */
3     return n + (1 << c0) + (1 << (c1 - 1)) - 1;
4 }
```

Арифметическое решение для getPrev

Если $c1$ — количество конечных единиц, $c0$ — размер блока соседних с ними нулей и $p = c0 + c1$, можно реализовать getPrev так:

Установите бит p в 0.

Установите все биты после p в 1.

Установите биты с 0 по $c0 - 1$ в 0.

Можно решить эту задачу арифметически. Предположим, что $n = 10000011$. Тогда $c1 = 2$, $c0 = 5$.

```

n -= 2c1 - 1;      // Удалить завершающие единицы. p теперь 10000000
n -= 1;           // Обратить завершающие нули. p теперь 01111111
n -= 2c0 - 1 - 1; // Обратить последние (c0-1) нули. p теперь 01110000
```

Это выражение можно сократить:

```

next = n - (2c1 - 1) - 1 - (2c0 - 1 - 1)
      = n - 2c1 - 2c0 - 1 + 1
```

Вновь получаем очень простое решение:

```
1 int getPrevArith(int n) {
2     /* ... то же вычисление с0 c1, как и раньше ... */
3     return n - (1 << c1) - (1 << (c0 - 1)) + 1;
4 }
```

Не волнуйтесь! Вам не придется выполнять такие задания на собеседовании (по крайней мере, без небольшой помощи интервьюера).

5.4. Объясните, что делает код: $((n \& (n-1)) == 0)$.

Решение

Вернемся к «истокам».

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 общих единиц.

На что похоже $n-1$ (по сравнению с n)?

Попытайтесь проделать вычитание вручную (в двоичной или десятичной системах). Что произойдет?

$$\begin{array}{rcl} 1101011000 & [\text{основа 2}] & 593100 & [\text{основа 10}] \\ - & 1 & - & 1 \\ \hline = 1101010111 & [\text{основа 2}] & = 593099 & [\text{основа 10}] \end{array}$$

Когда вы отнимаете единицу, посмотрите на младший бит. 1 вы замените на 0. Но, если там стоит 0, то вы должны заимствовать из старшего бита. Вы изменяете каждый бит с 0 на 1, пока 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 общих единиц. Предположим, они имеют вид:

- $n = abcde1000$
- $n-1 = abcde0111$

abcde должны быть нулевыми битами, то есть n имеет вид: 000001000. Таким образом, значение n — степень двойки.

Итак, наш ответ: логическое выражение $((n \& (n-1)) == 0)$ истинно, если n является степенью двойки или равно нулю.

- 5.5.** Напишите функцию, определяющую количество битов, которые необходимо изменить, чтобы из целого числа A получить целое число B.

Решение

На первый взгляд кажется, что задача сложная, но фактически она очень проста. Чтобы решить ее, задайте себе вопрос: «Как узнать, какие биты в двух числах различаются?». Ответ прост — с помощью операции XOR.

Каждая единица результирующего числа соответствует биту, который не совпадает в числах A и B. Поэтому расчет количества несовпадающих битов в числах A и B сводится к подсчету числа единиц в числе $A \wedge B$:

```
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 хорош, но можно сделать его еще лучше. Вместо многократного сдвига для проверки значащего бита достаточно будет инвертировать младший ненулевой разряд и подсчитывать, сколько раз понадобится проделать эту операцию, пока число не станет равным нулю. Операция $c = c \& (c - 1)$ очищает младший ненулевой бит числа c.

Приведенный далее 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 любят давать на собеседованиях. Если вы никогда с ними не сталкивались, вам будет сложно сразу решить задачу, поэтому запомните использованные здесь трюки, они пригодятся на собеседовании.

- 5.6.** Напишите программу, меняющую местами четные и нечетные биты числа.

Количество инструкций должно быть наименьшим (нужно поменять местами биты 0 и 1, 2 и 3 и т. д.).

Решение

Как и в предыдущих случаях, полезно взглянуть на задачу с другой стороны. Работа над парами битов по отдельности будет слишком сложной и не очень эффективной. Что можно сделать?

Можно решить задачу, выполняя сначала операции с нечетными, а потом с четными битами? Можно взять n и сдвинуть нечетные биты на 1? Да. Давайте сделаем 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:

```
1 public int swapOddEvenBits(int x) {
2     return ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
3 }
```

Это решение на Java для 32-разрядных чисел. Если вы работаете с 64-разрядными числами, нужно изменить маску, но логика останется той же.

5.7. Массив $A[1\dots n]$ содержит целые числа от 0 до n , но одно число отсутствует.

В этой задаче мы не можем получить доступ к любому числу в массиве A с помощью одной операции. Элементы массива A хранятся в двоичном виде, а доступ к ним осуществляется только при помощи команды извлечь j -й бит из $A[i]$, имеющей фиксированное время выполнения. Напишите код, обнаруживающий отсутствующее целое число. Можно ли выполнить эту задачу за время $O(n)$?

Решение

Попробуем решить сходную задачу: 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.

Время выполнения этой программы будет пропорционально $n * \text{length}(n)$, где length — это количество битов в n . Обратите внимание, что $\text{length}(n) = \log_2(n)$. Итак, время выполнения будет $O(n \log(n))$. Не слишком оптимистично.

Существуют ли другие способы решить эту задачу?

Можно использовать тот же подход, но работать 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 как LSB_1 . В списке чисел от 0 до n мы ожидаем получить одинаковое количество нулей и единиц (если n — нечетное) или дополнительный 0, если n — четное. То есть:

- если $n \% 2 == 1$, то $\text{count}(0s) = \text{count}(1s)$;
- если $n \% 2 == 0$, то $\text{count}(0s) = 1 + \text{count}(1s)$.

Это означает, что $\text{count}(0s)$ всегда будет больше или равно $\text{count}(1s)$.

Когда мы удаляем значение v из списка, чтобы узнать, является ли v четным или нечетным, достаточно взглянуть на младший бит остальных чисел в списке.

	$n \% 2 == 0$ $\text{count}(0s) = 1 + \text{count}(1s)$	$n \% 2 == 1$ $\text{count}(0s) = \text{count}(1s)$
$v \% 2 == 0$ $\text{LSB}_1(v) = 0$	0 удален. $\text{count}(0s) = \text{count}(1s)$	0 удален. $\text{count}(0s) < \text{count}(1s)$
$v \% 2 == 1$ $\text{LSB}_1(v) = 1$	1 удалена. $\text{count}(0s) > \text{count}(1s)$	1 удалена. $\text{count}(0s) > \text{count}(1s)$

То есть:

- если $\text{count}(0s) \leq \text{count}(1s)$, то v — четное;
- если $\text{count}(0s) > \text{count}(1s)$, то v — нечетное.

Но как узнать следующий бит v ? Если v находилось бы в списке, то следующий бит был бы:

$$\text{count}_2(0s) = 1 + \text{count}_2(1s) \text{ OR } \text{count}_2(0s) = 1 + \text{count}_2(1s),$$

где count_2 указывает на число 0 или 1 в LSB_2 .

Как в предыдущем примере, можно узнать значение второго младшего бита v :

	$\text{count}_2(0s) = 1 + \text{count}_2(1s)$	$\text{count}_2(0s) = \text{count}_2(1s)$
$\text{LSB}_2(v) == 0$	0 удален $\text{count}_2(0s) = \text{count}_2(1s)$	0 удален $\text{count}_2(0s) < \text{count}_2(1s)$
$\text{LSB}_2(v) == 1$	1 удалена. $\text{count}_2(0s) > \text{count}_2(1s)$	1 удалена. $\text{count}_2(0s) > \text{count}_2(1s)$

То есть:

- если $\text{count}_2(0s) \leq \text{count}_2(1s)$, то $\text{LSB}_2(v) = 0$;
- если $\text{count}_2(0s) > \text{count}_2(1s)$, то $\text{LSB}_2(v) = 1$.

Можно повторять этот процесс для каждого бита. На каждом шаге мы подсчитываем количество 0 и 1 в v , чтобы узнать, является $\text{LSB}_1(v)$ нулем или единицей. Затем мы отбрасываем значения, для которых $\text{LSB}_1(x) \neq \text{LSB}_1(v)$. 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 смотрим на биты n , затем на $n/2$, затем на $n/4$ и т. д. Время выполнения — $O(N)$.

Давайте визуализируем пример. На первой итерации мы начинаем с чисел:

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Поскольку $\text{count}_1(0s) > \text{count}_1(1s)$, то $\text{LSB}_1(v) = 1$. Отбросим все x , где $\text{LSB}_1(x) \neq \text{LSB}_1(v)$.

00000-	00100-	01000-	01100-
00001	00101	01001	01101
00010-	00110-	01010-	
-----	00111	01011	

Теперь $\text{count}_2(0s) > \text{count}_2(1s)$, мы знаем, что $\text{LSB}_2(v) = 1$. Отбросим все x , где $\text{LSB}_2(x) \neq \text{LSB}_2(v)$.

00000-	00100-	01000-	01100-
00001	00101	01001	01101
00010-	00110-	01010-	
-----	00111	01011	

На этот раз $\text{count}_3(0s) \leq \text{count}_3(1s)$. Мы знаем, что $\text{LSB}_3(v) = 0$. Отбросим все x , где $\text{LSB}_3(x) \neq \text{LSB}_3(v)$.

00000	00100	01000	01100
00001	00101	01001	01101
00010	00110	01010	
-----	00111	01011	

Осталось единственное число. В этом случае $\text{count}_4(0s) \leq \text{count}_4(1s)$, поэтому $\text{LSB}_4(v) = 0$.

Когда все числа, где $\text{LSB}_4(v) \neq 0$, отброшены, список оказывается пустым. Как только список пуст, $\text{count}_1(0s) \leq \text{count}_1(1s)$. Итак, $\text{LSB}_1(v) = 0$. Получив пустой список, мы можем заполнить оставшиеся биты v нулями.

Этот процесс позволяет найти значение v . Для нашего случая получаем $v = 00011$.

Представленный далее код реализует наш алгоритм.

```

1 public int findMissing(ArrayList<BitInteger> array) {
2     /* BitInteger.INTEGER_SIZE - 1 соответствует LSB. Начнем
3      * отсюда, и пройдемся по большим битам*/
4     return findMissing(array, BitInteger.INTEGER_SIZE - 1);
5 }
6
7 public int findMissing(ArrayList<BitInteger> input, int column) {
8     if (column < 0) { // Базовый случай и условие ошибки
9         return 0;
10    }
11    ArrayList<BitInteger> oneBits =
12        new ArrayList<BitInteger>(input.size()/2);
13    ArrayList<BitInteger> zeroBits =
14        new ArrayList<BitInteger>(input.size()/2);
15
16    for (BitInteger t : input) {
17        if (t.fetch(column) == 0) {
18            zeroBits.add(t);
19        } else {
20            oneBits.add(t);
21        }
22    }
23    if (zeroBits.size() <= oneBits.size()) {
24        int v = findMissing(zeroBits, column - 1);
25        return (v << 1) | 0;
26    } else {
27        int v = findMissing(oneBits, column - 1);
28        return (v << 1) | 1;
29    }
30 }
```

В строках с 24 по 27 рекурсивно рассчитываются биты v . 0 или 1 выбираются в зависимости от того, истинно ли условие $\text{count}_1(0s) \leq \text{count}_1(1s)$.

- 5.8. Изображение с монохромного экрана сохранено как одномерный массив байтов, так что в одном байте хранится информация о восьми соседних пикселях. Ширина изображения в кратна 8 (байты соответствуют столбцам). Высоту экрана можно рассчитать, зная длину массива и ширину экрана. Реализуйте функцию `drawHorizontalLine(byte[] screen, int width, int x1, int x2, int y)`, которая рисует горизонтальную линию из точки (x_1, y) в точку (x_2, y) .

Решение

Действовать прямолинейно — использовать цикл и двигаться от x_1 до x_2 , расставляя каждый пиксель, — довольно сложный и не очень эффективный метод.

Лучшее решение — провести анализ ситуации и прийти к выводу, что если x_1 и x_2 находятся далеко друг от друга, то между ними 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    // Устанавливаем полные байты  
15    for (int b = first_full_byte; b <= last_full_byte; b++) {  
16        screen[(width / 8) * y + b] = (byte) 0xFF;  
17    }  
18  
19    // Создаем маски для начала и конца строки  
20    byte start_mask = (byte) (0xFF >> start_offset);  
21    byte end_mask = (byte) ~(0xFF >> (end_offset + 1));  
22  
23    // Устанавливаем начало и конец строки  
24    if ((x1 / 8) == (x2 / 8)) { // x1 and x2 are in the same byte  
25        byte mask = (byte) (start_mask & end_mask);  
26        screen[(width / 8) * y + first_full_byte - 1] |= 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; //распоряжения
34         screen[byte_number] |= end_mask;
35     }
36 }
37 }
```

Будьте осторожны при решении подобной задачи: нужно рассмотреть все «специальные» случаи. Предусмотрите ситуацию, когда x_1 и x_2 — один байт. Только самые внимательные кандидаты могут реализовать этот код без ошибок.

6. Головоломки

- 6.1.** Дано: 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 «хитрые» ограничения могут стать подсказкой. В нашем случае подсказка спрятана в информации о том, что весы можно использовать только один раз.

У нас только одно взвешивание, а это значит, что придется одновременно взвешивать много таблеток. Фактически, мы должны одновременно взвесить все 19 банок. Если мы пропустим две (или больше) банки, то не сможем их проверить. Не забывайте: только одно взвешивание!

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. Подход к решению задачи найден.

Мы знаем ожидаемый вес таблеток. Если мы возьмем разное количество таблеток из каждой банки, то разница между ожидаемым и фактическим весом покажет, какая банка содержит более тяжелые таблетки.

Можно обобщить наш подход: 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. «Излишек» внесет банка с тяжелыми таблетками.

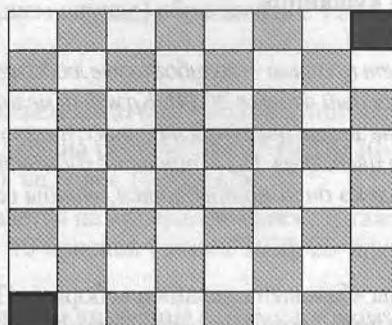
Таким образом, номер банки можно узнать по простой формуле: $(вес - 210) / 0,1$. Если суммарный вес таблеток составляет 211,3 г, то тяжелые таблетки находились в банке № 13.

- 6.2.** Дано: 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 свой ответ.

Решение

С первого взгляда кажется, что это возможно. Доска 8×8, следовательно, есть 64 клетки, две мы исключаем, значит, остается 62. Вроде бы 31 кость должна поместиться, правильно?

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, у нас никогда не получится разложить все кости.

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 клетки. Но на нашей доске всего 30 черных и 32 белые клетки. Поэтому разложить кости невозможно.

- 6.3.** У вас есть пятилитровый и трехлитровый кувшины и неограниченное количество воды. Как отмерить ровно 4 литра? Кувшины имеют неправильную форму, поэтому точно отмерить «половину» кувшина невозможно.

Решение

Если мы поэкспериментируем с кувшинами, то обнаружим, что можно отмерить 4 литра, если переливать воду из кувшина в кувшин следующим образом:

5 л	3 л	Примечание
5	0	Заполнили 5-литровый кувшин.
2	3	Из 5-литрового перелили в 3-литровый. В 5-литровом и 3-литровом кувшинах всего 5 литров воды
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

Обратите внимание, что у многих головоломок есть математическая основа. Если размеры двух кувшинов являются единственным ограничением, то вы сможете найти последовательность переливаний воды для любого значения объема между единицей и суммарным объемом кувшинов.

- 6.4.** На острове существует правило — голубоглазые люди не могут там находиться. Самолет улетает каждый вечер в 20:00. Каждый человек может видеть цвет глаз других людей, но не знает цвет собственных, никто не имеет права сказать человеку, какой у него цвет глаз. На острове находится не менее одного голубоглазого человека. Сколько дней потребуется, чтобы все голубоглазые уехали?

Решение

Давайте используем подходы «базовый случай» и «сборка». Предположим, что на острове находится n людей и c из них — голубоглазые. Таким образом, мы знаем, 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 один 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 голубоглаз. Оба человека должны будут вечером покинуть остров.

$c > 2$: общий случай

Давайте использовать ту же логику. Если $c = 3$, то эти три человека сразу увидят, что на острове есть еще 2 (или 3) 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.

Такой шаблон можно использовать для произвольного значения c . Поэтому если на острове находится c человек с голубыми глазами, понадобится c ночей, чтобы все они покинули остров.

- 6.5.** Дано 100-этажное здание. Если яйцо сбросить с высоты N -го этажа (или с большей высоты), оно разбьется. Если его бросить с любого меньшего этажа, оно не разбьется. У вас есть два яйца, найдите N за минимальное количество бросков.

Решение

Обратите внимание, что независимо от того, с какого этажа мы бросаем яйцо № 1, бросая яйцо № 2, необходимо использовать линейный поиск (от самого низкого до

самого высокого этажа) между этажом «повреждения» 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.

Подход

Предположим, что мы бросаем яйцо с 10-го этажа, потом с 20-го...

- Если яйцо № 1 разбилось на первом броске (этаж 10-й), то нам в худшем случае придется проделать не более 10 бросков.
- Если яйцо № 1 разбивается на последнем броске (этаж 100-й), тогда у нас впереди в худшем случае еще 19 бросков (этажи 10-й, 20-й, ..., 90-й, 100-й, затем с 91-го до 99-го).

Это хорошо, но давайте уделим внимание самому плохому случаю. Выполним балансировку нагрузки, чтобы выделить два наиболее вероятных случая.

1. В хорошо сбалансированной системе значение $Drops(Egg1) + Drops(Egg2)$ будет постоянным, независимо от того, на каком этаже разбилось яйцо № 1.
2. Допустим, что за каждый бросок яйцо № 1 «делает» один шаг (этаж), а яйцо № 2 перемещается на один шаг меньше.
3. Нужно каждый раз сокращать на единицу 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 снизить количество бросков яйца № 2 до 8. Для этого достаточно бросить яйцо № 1 с 39-го этажа.
4. Мы знаем, что яйцо № 1 должно стартовать с этажа X , затем спуститься на $X-1$ этажей, затем — на $X-2$ этажей, пока не будет достигнуто число 100.
5. Можно вывести формулу, описывающую наше решение: $X + (X-1) + (X-2) + \dots + 1 = 100 \rightarrow X = 14$.

Таким образом, мы сначала попадаем на 14-й этаж, затем на 27-й, затем на 39-й. Так что 14 шагов — худший случай.

Как и в других задачах максимизации/минимизации; ключом к решению является «балансировка худшего случая».

- 6.6. Дано:** 100 закрытых замков расположены в длинном коридоре. Человек сначала открывает все 100. 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). На 100-м проходе человек должен «переключить» только замок № 100. Сколько замков остались открытыми?

Решение

Давайте сначала определимся с тем, что такое «переключение». Это поможет нам разобраться, какие замки останутся открытыми.

Вопрос: на каком проходе происходит переключение (открытие или закрытие)?

Замок n переключается один раз для каждого значения n (включая n и 1). Таким образом, замок № 15 переключается на 1, 3, 5 и 15 проходах.

Вопрос: когда замок открыт?

Замок открыт, если значение счетчика (назовем его x) нечетное. Это можно использовать, разделив коэффициенты на «открытые» и «закрытые».

Вопрос: когда x будет нечетным?

Значение x будет нечетным, если n — квадрат числа. 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 коэффициент, таким образом, мы получаем нечетное количество коэффициентов.

Вопрос: сколько квадратов в нашей задаче?

В этой задаче 10 квадратов — $(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)$ — вы можете взять коэффициенты от 1 до 10 и возвести их в квадрат:

$$1^2, 2^2, 3^2, \dots, 10^2$$

Таким образом, после всех обходов 10 замков останутся открытыми.

7. Математика и теория вероятностей

- 7.1. Вам предлагают бросить мяч в баскетбольное кольцо. Существует два варианта игры:

Вариант 1: попасть в кольцо с одного броска.

Вариант 2: у вас есть три попытки, и нужно попасть в кольцо два раза из трех.

Если p – вероятность попадания, то как, в зависимости от значения p , выбрать более выигрышный вариант игры?

Решение

Для решения этой задачи можно напрямую использовать законы теории вероятностей, чтобы сравнить вероятность победы в каждом случае.

Вероятность победы в игре 1.

Вероятность победы в игре составит $1 - p$.

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 две и более будут удачными. Другими словами:

$$P(\text{победа}) = s(2, 3) + s(3, 3)$$

Вероятность трех попаданий:

$$s(3, 3) = p^3$$

Вероятность двух удачных попаданий:

$$\begin{aligned} P(\text{попадание 1 и 2, промах - 3}) &+ \\ &+ P(\text{попадание 1 и 3, промах 2}) + \\ &+ P(\text{промах 1, попадание 2 и 3}) = \\ &= p * p * (1 - p) + p * (1 - p) * p + (1 - p) * p * p \\ &= 3 * (1 - p) * p^2 \end{aligned}$$

Собрав все вместе, мы получим:

$$\begin{aligned} &= p^3 + 3 * (1 - p) * p^2 \\ &= p^3 + 3p^2 - 3p^3 \\ &= 3p^2 - 2p^3 \end{aligned}$$

Какой вариант игры выбрать?

Вам нужно играть по правилам игры 1, если $P(\text{игра 1}) > P(\text{игра 2})$:

$$p > 3p^2 - 2p^3.$$

$$1 > 3p - 2p^2$$

$$2p^2 - 3p + 1 > 0$$

$$(2p - 1)(p - 1) > 0$$

Оба значения могут быть либо положительными, либо отрицательными. Но мы знаем, что $p < 1$, следовательно, $p - 1 < 0$. Это говорит о том, что оба значения должны быть отрицательными.

$$2p - 1 < 0$$

$$2p < 1$$

$$p < .5$$

Итак, нам нужно играть в игру 1, если $p < .5$. Если же $p = 0, 0.5$ или 1 , значит, $P(\text{игра } 1) = P(\text{игра } 2)$, поэтому не имеет значения, в какую игру играть.

- 7.2.** Три муравья находятся в вершинах треугольника. Какова вероятность столкновения между какими-либо двумя или всеми тремя муравьями, если муравьи начинают двигаться вдоль сторон треугольника? (Муравей выбирает любое направление с равной вероятностью, скорость движения муравьев одинакова.)

Найдите вероятность столкновения для случая, когда n муравьев находятся в вершинах n -угольника.

Решение

Единственный способ избежать столкновений — двигаться в одном направлении (по часовой или против часовой стрелки). Мы можем рассчитать эту вероятность, решая задачу в «обратном» направлении.

Так как каждый муравей может двигаться в двух направлениях, всего у нас три муравья, вероятность составляет:

$$P(\text{по часовой стрелке}) = (\frac{1}{2})^3$$

$$P(\text{против часовой стрелки}) = (\frac{1}{2})^3$$

$$P(\text{в одном направлении}) = (\frac{1}{2})^3 + (\frac{1}{2})^3 = \frac{1}{2}$$

Вероятность столкновения муравьев теперь можно представить как вероятность того, что муравьи не будут двигаться в одном направлении:

$$P(\text{столкновение}) = 1 - P(\text{в том же направлении}) = 1 - \frac{1}{2} = \frac{1}{2}$$

Обобщим это решение для случая n -многоугольника: существует только два маршрута, по которым муравьи могут двигаться без столкновений, но возможных маршрутов теперь 2^n . Общая вероятность столкновения:

$$P(\text{по часовой стрелке}) = (\frac{1}{2})^n$$

$$P(\text{против часовой стрелки}) = (\frac{1}{2})^n$$

$$P(\text{в одном направлении}) = 2(\frac{1}{2})^n (\frac{1}{2})^n = (\frac{1}{2})^{n-1}$$

$$P(\text{столкновение}) = 1 - P(\text{в одном направлении}) = 1 - (\frac{1}{2})^{n-1}$$

- 7.3.** Какова вероятность пересечения двух прямых, лежащих в одной плоскости?

Решение

Задача содержит много неопределенностей:

В каком формате заданы прямые?

Могут ли прямые совпадать?

Все эти вопросы нужно обсудить с интервьюером.

Допустим, что нам необходимо разработать структуру данных для хранения информации о прямой, и будем считать, что если две линии совпадают, то они пересекаются.

Надеюсь, что из курса школы вы помните, что если две линии, лежащие в одной плоскости, не параллельны, то они пересекаются. Таким образом, чтобы проверить, пересекаются ли две линии, достаточно проверить, различаются ли их наклоны.

Такую задачу можно реализовать с помощью следующего кода:

```
1 public class Line {  
2     static double epsilon = 0.000001;  
3     public double slope;  
4     public double yintercept;  
5  
6     public Line(double s, double y) {  
7         slope = s;  
8         yintercept = y;  
9     }  
10  
11    public boolean intersect(Line line2) {  
12        return Math.abs(slope - line2.slope) > epsilon ||  
13            Math.abs(yintercept - line2.yintercept) < epsilon;  
14    }  
15 }
```

При решении задач такого типа нужно помнить о следующем:

- Задавайте вопросы, ведь в таких задачах много неоднозначностей. Многие интервьюеры преднамеренно добавляют «белые пятна», чтобы посмотреть, как вы будете действовать.
- При необходимости создавайте и используйте собственные структуры данных. Это показывает, что вы способны создать объектно-ориентированный проект.
- Подумайте, какие структуры данных подходят для представления линий. Существует множество вариантов. Обоснуйте свой выбор.
- Не в коем случае не упрощайте задачу, предполагая, что наклон и сдвиг прямой — целые числа.
- Не забывайте про ограничения представлений чисел с плавающей точкой. Не используйте проверку на равенство (`==`), вместо этого сравнивайте разницу между величинами со значением `epsilon`.

7.4. Используя только оператор суммирования, напишите методы, реализующие операции умножения, вычитания и деления целых чисел.

Решение

Нам разрешили использовать только сложение. В подобных задачах полезно вспомнить суть математических операций и как их можно реализовать с помощью сложения (или других операций).

Вычитание

Как реализовать вычитание с помощью сложения? Это предельно просто. Операция $a - b$ — то же самое, что и $a + (-1) \times b$. Поскольку мы не можем использовать оператор умножения, нам придется создать функцию *negate*.

```

1  /* Меняем знак числа */
2  public static int negate(int a) {
3      int neg = 0;
4      int d = a < 0 ? 1 : -1;
5      while (a != 0) {
6          neg += d;
7          a += d;
8      }
9      return neg;
10 }
11
12 /* Вычитание реализуется путем изменения знака числа b и суммирования чисел a + b */
13 public static int minus(int a, int b) {
14     return a + negate(b);
15 }
```

Отрицательное значение *k* получается суммированием *k* раз числа -1 .

Умножение

Связь между сложением и умножением тоже достаточно очевидна. Чтобы перемножить *a* и *b*, нужно сложить значение *a* с самим собой *b* раз.

```

1  /* Перемножение a и b путем суммирования */
2  public static int multiply(int a, int b) {
3      if (a < b) {
4          return multiply(b, a); // алгоритм будет быстрее, если b < a
5      }
6      int sum = 0;
7      for (int i = abs(b); i > 0; i--) {
8          sum += a;
9      }
10     if (b < 0) {
11         sum = negate(sum);
12     }
13     return sum;
14 }
15
16 /* Возвращаем абсолютное значение */
17 public static int abs(int a) {
18     if (a < 0) {
19         return negate(a);
20     } else {
21         return a;
22     }
23 }
```

При умножении нам нужно обратить особое внимание на отрицательные числа. Если b — отрицательное число, то необходимо учесть знак суммы:

```
multiply(a, b) <- abs(b) * a * (-1 if b < 0).
```

Кроме того, для решения этой задачи мы создали простую функцию `abs`.

Деление

Самая сложная из математических операций — деление. Хорошая идея — использовать для реализации метода `divide` методы `multiply`, `subtract` и `negate`.

Нам нужно найти x , если $x = a / b$. Давайте переформулируем задачу: найти x , если $a = bx$. 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.

Взгляните еще раз на уравнение $a = xb$. Обратите внимание, что можно вычислить x как результат суммирования b , пока не будет получено a . Количество экземпляров b , необходимых, чтобы получить a , и будет искомой величиной x .

Конечно, это решение нельзя назвать делением, но оно работает. Нас попросили реализовать целочисленные операции, с чем мы и справились. Хотя вы должны понимать, что при такой реализации не получить остаток от деления.

Приведенный ниже код реализует данный алгоритм:

```
1 public int divide(int a, int b)
2 throws java.lang.ArithmetricException {
3     if (b == 0) {
4         throw new java.lang.ArithmetricException("ERROR");
5     }
6     int absa = abs(a);
7     int absb = abs(b);
8     int product = 0;
9     int x = 0;
10    while (product + absb <= absa) { /* пока не достигнем a */
11        product += absb;
12        x++;
13    }
14    if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
15        return x;
16    } else {
17        return negate(x);
18    }
19}
20}
21}
```

При решении этой задачи вы должны помнить о следующем:

- Подойдите к решению логическим путем, вспомните, что делает умножение и деление. Все задачи собеседования имеют логичное решение.

- Вы имеете шанс продемонстрировать свою способность написать чистый и понятный код, пригодный для повторного использования. Если вы решили эту задачу самостоятельно, но не выделили `negate` в отдельный метод, то не забудьте на собеседовании создать дополнительный метод, который сделает проще повторное использование вашей программы.
- Будьте осторожны при использовании «упрощающих» задачу дополнительных условий. Не нужно предполагать, что все числа положительны или что $a > b$.

7.5. Для двух квадратов, лежащих в одной плоскости, найдите прямую, которая делала бы эти квадраты пополам. (Стороны квадратов параллельны осям координат.)

Решение

Прежде чем начать, нам нужно подумать, что представляет собой объект «прямая». Прямая будет задаваться наклоном и сдвигом? Или какими-либо двумя точками? Или прямая на самом деле является отрезком, начало и конец которого лежат на сторонах квадратов?

Давайте сделаем задачу более интересной и рассмотрим третий вариант — возьмем отрезок, начало и конец которого лежат на сторонах квадратов. Не забудьте обсудить данный момент с интервьюером.

Прямая, которая делит два квадрата пополам, должна проходить через их середины. Наклон прямой описывается формулой: $slope = (y_1 - y_2) / (x_1 - x_2)$. Этим же принципом можно руководствоваться, чтобы рассчитать начальную и конечную точки отрезка.

```

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     /* Возвращаем точку, где прямая, соединяющая mid1 и mid2,
9      * пересекает сторону квадрата 1. Таким образом, рисуем прямую,
10     * из точки mid2 в точку mid1 и продолжаем до стороны
11     * квадрата.
12     */
13    public Point extend(Point mid1, Point mid2, double size) {
14        /* прямая пересекает сторону квадрата, которая расположена на
15         * расстоянии (size/2) правее, левее, выше или ниже центра квадрата
16         * Если mid1 левее mid2, то прямая пересекает сторону квадрата
17         * левее mid1. Если mid1 больше mid2, то прямая пересекает сторону
18         * квадрата выше mid1.
19         */
20        double xdir = mid1.x < mid2.x ? -1 : 1;
21        double ydir = mid1.y < mid2.y ? -1 : 1;
22
23        /* Если у mid1 и mid2 одинаковая координата x, то вычисление
24         * наклона приведет к делению на 0. Чтобы избежать этого,
25         * сделаем дополнительную обработку, так как нам известно, что

```

```

26     * в этом случае точки пересечения будут иметь ту же x-координату.
27     */
28     if (mid1.x == mid2.x) {
29         return new Point(mid1.x, mid1.y + ydir * size / 2.0);
30     }
31
32     double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
33     double x1 = 0;
34     double y1 = 0;
35
36     /* Вычисляем x1 и y1, используя уравнение:
37      * slope = (y1 - y2) / (x1 - x2).
38      * Если наклон крутой, конец отрезка линии
39      * пересечет горизонтальную сторону квадрата.
40      * Если наклон пологий, конец отрезка линии
41      * пересечет вертикальную сторону квадрата
42      *
43      */
44     if (Math.abs(slope) == 1) {
45         x1 = mid1.x + xdir * size / 2.0;
46         y1 = mid1.y + ydir * size / 2.0;
47     } else if (Math.abs(slope) < 1) { // пологий наклон
48         x1 = mid1.x + xdir * size / 2.0;
49         y1 = slope * (x1 - mid1.x) + mid1.y;
50     } else { // крутой наклон
51         y1 = mid1.y + ydir * size / 2.0;
52         x1 = (y1 - mid1.y) / slope + mid1.x;
53     }
54     return new Point(x1, y1);
55 }
56
57 public Line cut(Square other) {
58     Point middle_s = this.middle();
59     Point middle_t = other.middle();
60     if (middle_s.isEqual(middle_t)) {
61         Square bigger = bottom - top > other.bottom - other.top ?
62             this : other;
63         return new Line(new Point(bigger.left, bigger.top),
64                         new Point(bigger.right, bigger.bottom));
65     } else {
66         Point point_s = extend(middle_s, middle_t,
67             this.right - this.left);
68         Point point_t = extend(middle_t, middle_s,
69             other.right - other.left);
70     return new Line(point_s, point_t); }
71 }
```

Основная цель этой задачи — увидеть, насколько вы внимательно относитесь к написанию кода. Достаточно взглянуть на особые случаи (когда середины квадратов совпадают по какой-либо из осей). Вы должны продумать поведение программы

в специальных случаях прежде, чем начнете писать код. Убедитесь, что надлежащим образом обработали сложные ситуации.

7.6. Дано: набор точек на плоскости. Найдите прямую линию, которая проходит через большинство точек.

Решение

Если мы соединим между собой все возможные пары точек, то определим, какая прямая является наиболее распространенной (встречается большее количество раз). Если решать задачу «в лоб», то чтобы узнать, сколько точек удовлетворяют нашему условию, придется выполнять проверку каждого отрезка. Это потребует $O(N^3)$ времени, поскольку необходимо проанализировать N^2 точек и перебрать $O(N)$ отрезков.

Прежде чем приступить к поиску оптимального решения, давайте обсудим варианты представления прямой линии:

- Пара точек. В этом случае мы описываем одну и ту же прямую бесконечным числом способов.
- Наклон и сдвиг. Преимущество второго подхода в том, что любую линию можно однозначно идентифицировать по наклону и сдвигу.

Теперь давайте обдумаем наш выбор. У нас есть множество прямых, представленных как наклон и сдвиг, и мы хотим найти наиболее часто встречающуюся комбинацию наклона и сдвига. Как мы можем это сделать?

На самом деле задача не очень отличается от классической задачи «найти наиболее часто встречающееся число в списке чисел». Нам нужно всего лишь пройтись по всем прямым и использовать хэш-таблицу для подсчета количества повторов.

```

1 public static Line findBestLine(GraphPoint[] points) {
2     Line bestLine = null;
3     HashMap<Line, Integer> line_count =
4         new HashMap<Line, Integer>();
5     for (int i = 0; i < points.length; i++) {
6         for (int j = i + 1; j < points.length; j++) {
7             Line line = new Line(points[i], points[j]);
8             if (!line_count.containsKey(line)) {
9                 line_count.put(line, 0);
10            }
11            line_count.put(line, line_count.get(line) + 1);
12            if (bestLine == null ||
13                line_count.get(line) > line_count.get(bestLine)) {
14                bestLine = line;
15            }
16        }
17    }
18    return bestLine;
19 }
20
21 public class Line {
22     private static double epsilon = .0001;

```

```

23     public double slope;
24     public double intercept;
25     private boolean infinite_slope = false;
26     public Line(GraphPoint p, GraphPoint q) {
27         if (Math.abs(p.x - q.x) > epsilon) { // если координаты x отличаются
28             slope = (p.y - q.y) / (p.x - q.x); // вычисляем наклон
29             intercept = p.y - slope * p.x; // y-пересечение из y=mx+b
30         } else {
31             infinite_slope = true;
32             intercept = p.x; // x-пересечение, так как наклон бесконечен
33         }
34     }
35
36     public boolean isEqual(double a, double b) {
37         return (Math.abs(a - b) < epsilon);
38     }
39
40     @Override
41     public int hashCode() {
42         int sl = (int)(slope * 1000);
43         int in = (int)(intercept * 1000);
44         return sl | in;
45     }
46
47     @Override
48     public boolean equals(Object o) {
49         Line l = (Line) o;
50         if (isEqual(l.slope, slope) &&
51             isEqual(l.intercept, intercept) &&
52             infinite_slope == l.infinite_slope) {
53             return true;
54         }
55         return false;
56     }
57 }
```

Обратите внимание на расчет наклона линии. Линия может оказаться вертикальной. Этот случай можно отследить с помощью флага (`infinite_slope`) и проверки методом `equals`. Не забудьте, что деление не является точной операцией, поэтому проверку наклона прямых необходимо выполнять сравнением разницы этих двух величин со значением `epsilon`.

- 7.7. Разработайте алгоритм, позволяющий найти k -е число из упорядоченного числового ряда, в разложении элементов которого на простые множители присутствуют только 3, 5 и 7.

Решение

По условию задачи любое число этого ряда должно представлять собой произведение $3^a \cdot 5^b \cdot 7^c$.

Давайте посмотрим на список чисел, удовлетворяющих нашим требованиям.

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	$5*7$	$3^0 * 5^1 * 7^1$
45	$5*9$	$3^2 * 5^1 * 7^0$
49	$7*7$	$3^0 * 5^0 * 7^2$
63	$3*21$	$3^2 * 5^0 * 7^1$

Поскольку $3^{a-1} * 5^b * 7^c < 3^a * 5^b * 7^c$, то число $3^{a-1} * 5^b * 7^c$ должно попасть в список, как и все перечисленные далее числа:

$$3^{a-1} * 5 * 7^c$$

$$3^a * 5^{b-1} * 7^c$$

$$3^a * 5^b * 7^{c-1}$$

Другой способ — представить каждое число в следующем виде:

- $3 * (\text{некоторое предыдущее число из числового ряда});$
- $5 * (\text{некоторое предыдущее число из числового ряда});$
- $7 * (\text{некоторое предыдущее число из числового ряда}).$

Мы знаем, что A_k можно записать как $(3, 5 \text{ или } 7) * (\text{некоторое значение из } \{A_1, \dots, A_{k-1}\})$. Мы также знаем, что A_k является следующим числом в данном ряду. Поэтому A_k должно быть наименьшим «новым» числом, которое может быть получено умножением каждого значения в списке на 3, 5 или 7.

Как найти A_k ? Мы можем умножить каждое число в списке на 3, 5 или 7 и найти наименьший новый результат. Но такое решение потребует $O(k^2)$ времени. Неплохо, но можно сделать и лучше.

Вместо умножения всех элементов списка на 3, 5 или 7, можно рассматривать каждое предыдущее значение как основу для расчета трех последующих значений. Таким образом, каждое число A_1 может использоваться для формирования следующих форм:

$$3 * A_1$$

$$5 * A_1$$

$$7 * A_1$$

Эта идея поможет нам спланировать все заранее. Каждый раз, когда мы добавляем в список число A_1 , мы держим значения $3A_1$, $5A_1$, и $7A_1$ в «резервном» списке. Чтобы получить A_{i+1} , достаточно будет найти наименьшее значение во временном списке.

Наш код может быть таким:

```

1 public static 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 public static 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 public static int getKthMagicNumber(int k) {
21     if (k < 0) return 0;
22     Queue<Integer> q = new LinkedList<Integer>();
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 гораздо лучше предыдущего, но все еще не идеален.

Для генерирования нового элемента A_i мы осуществляем поиск по связному списку, где каждый элемент имеет вид:

- $3 * \text{предыдущий элемент}$;
- $5 * \text{предыдущий элемент}$;
- $7 * \text{предыдущий элемент}$.

Давайте уберем избыточные расчеты. Допустим, что наш 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 $7A_1 < \text{min}$, а затем $7A_5 < \text{min}$. Глупо, не правда ли? Поскольку мы знаем, что $A_1 < A_5$, то достаточно выполнить проверку $7A_1 < \text{min}$.

Если бы мы разделили список по постоянным множителям, то должны были бы проверить только первое из произведений на 3, 5 и 7. Все последующие элементы будут больше.

Таким образом, наш список принимает вид:

$$Q3 = \{3A_4\}$$

$$Q5 = \{5A_2, 5A_4, 5A_5\}$$

$$Q7 = \{7A_1, 7A_2, 7A_3, 7A_4, 7A_5\}$$

Чтобы найти минимум, достаточно проверить начало каждой очереди:

```
y = min(Q3.head(), Q5.head(), Q7.head())
```

Как только мы вычислим y , нужно добавить $3y$ в список $Q3$, $5y$ в $Q5$ и $7y$ в $Q7$. Но мы хотим вставлять эти элементы, только если они отсутствуют в других списках.

Как, например, $3y$ может попасть в какой-нибудь другой список? Допустим, элемент u был получен из $Q7$, это означает, что $u = 7x$. Если $7x$ — наименьшее значение, значит, $3x$ уже было задействовано. А как мы действовали, когда увидели $3x$? Правильно, мы вставили $7 * 3x$ в $Q7$. Обратите внимание, что $7 * 3x = 3 * 7x = 3y$.

В общем виде: если мы берем элемент из $Q7$, он будет иметь вид $7 * \text{suffix}$. Мы знаем, что $3 * \text{suffix}$ и $5 * \text{suffix}$ уже обработаны, и элемент $7 * 3 * \text{suffix}$ добавлен в $Q7$. При обработке $5 * \text{suffix}$ мы знаем, что $7 * 5 * \text{suffix}$ был добавлен в $Q7$. Единственное значение, которое мы еще не встречали, — $7 * 7 * \text{suffix}$, поэтому добавляем его в $Q7$.

Давайте рассмотрим пример, чтобы разобраться, как работает данный алгоритм:

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 $Q7$

$$Q3 = 3*3*3$$

$$Q5 = 5*5, 5*3*3, 5*5*3$$

$$Q7 = 7*5, 7*7, 7*3*3, 7*5*3, 7*7*3$$

Структура нашего алгоритма будет иметь вид:

1. Инициализируем `array` и очереди Q_3 , Q_5 и Q_7 .
2. Вставляем 1 в `array`.
3. Вставляем $1 \cdot 3$, $1 \cdot 5$ и $1 \cdot 7$ в Q_3 , Q_5 и Q_7 соответственно.
4. Пусть x будет минимальным элементом в Q_3 , Q_5 и Q_7 . Присоединим x к `magic`.
5. Если x находится в:
 - Q_3 → присоединяем $x \cdot 3$, $x \cdot 5$ и $x \cdot 7$ к Q_3 , Q_5 и Q_7 . Удаляем x из Q_3 .
 - Q_5 → присоединяем $x \cdot 5$ и $x \cdot 7$ к Q_5 и Q_7 . Удаляем x из Q_5 .
 - Q_7 → присоединяем $x \cdot 7$ только к Q_7 . Удаляем x из Q_7 .
6. Повторяем шаги 4–6, пока k -й элемент не будет найден.

Следующий код реализует данный алгоритм:

```

1 public static 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    /* Итерация от 0 до k */
12    for (int i = 0; i <= k; i++) {
13        int v3 = queue3.size() > 0 ? queue3.peek() :
14            Integer.MAX_VALUE;
15        int v5 = queue5.size() > 0 ? queue5.peek() :
16            Integer.MAX_VALUE;
17        int v7 = queue7.size() > 0 ? queue7.peek() :
18            Integer.MAX_VALUE;
19        val = Math.min(v3, Math.min(v5, v7));
20        if (val == v3) { // ставим в очередь 3, 5 и 7
21            queue3.remove();
22            queue3.add(3 * val);
23            queue5.add(5 * val);
24        } else if (val == v5) { // ставим в очередь 5 и 7
25            queue5.remove();
26            queue5.add(5 * val);
27        } else if (val == v7) { // ставим в очередь Q7
28            queue7.remove();
29        }
30        queue7.add(7 * val); // всегда добавляем в очередь Q7
31    }
32    return val;
33 }
```

Если вам досталась подобная задача, приложите все усилия, чтобы ее решить, потому что это действительно трудное задание. Вы можете начать с решения «в лоб» (спорно,

зато не слишком сложно), а затем попытаться оптимизировать его. Или попытайтесь найти шаблон, спрятанный в числах.

Интервьюер поможет, если вы будете испытывать затруднения. Не сдавайтесь! Рассуждайте вслух, задавайте вопросы и объясняйте ход ваших мыслей. Интервьюер на верняка начнет помогать вам.

Помните, никто не ожидает, что вы найдете идеальное решение. Ваши результаты будут сравнивать с результатами других кандидатов. Все будут находиться в одинаковых условиях.

Как, например, вы можете обнаружить, что в других случаях вам уже был получен изотип, это означает, что в этом конкретном случае уже было задействовано значение z для $z \in Q_7$. Следовательно, значение $7 * z$ в Q_7 . Обратите внимание, что в этом случае мы нашли точное значение z для $z \in Q_7$. В общем виде, если мы берем элемент $z \in Q_7$, он будет иметь изотип $7 * z$ для всех $z \in Q_7$, которые уже обработаны, и значение $7 * z$ будет $(z * z) * 7$. При обработке $5 * z$ мы знаем, что $7 * 5 * z$ либо было добавлено в базу знаний, либо имеет значение, которое мы еще не встречали — $7 * 5 * z$ будет, поэтому доказываем это в Давайте рассмотрим пример, чтобы разобраться, как работает этот алгоритм.

Возьмем изотип $z = 3$, вставленный в Q_3 , $5 * z$ в Q_5 , $7 * z$ в Q_7 .

$Q_3 = z$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 3$, вставлен изотип $5 * 3$ в Q_5 , $7 * 3$ в Q_7 . (* я од. 0 то я идишь *)

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5, 5 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7, 7 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 5, 5 * 3$ — пусть значит, мы уже обработали $5 * 3$ в Q_5 , $7 * 3$ в Q_7 :

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7, 7 * 3, 7 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 7, 7 * 3, 7 * 5$ — пусть значит, мы уже обработали $7 * 3$ в Q_7 , $7 * 5$ в Q_5 . Уже в π_1 осталось в инверте $\pi_1 = (\text{rev} * \text{book}) \vee \text{book}$:

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7 * 3, 7 * 5, 7 * 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 7 * 3, 7 * 5, 7 * 7$ — пусть значит, мы уже обработали $7 * 3$ в Q_7 , $7 * 5$ в Q_5 , $7 * 7$ в Q_3 :

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7 * 3, 7 * 5, 7 * 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 5 * 3, 5 * 5$ — пусть значит, мы уже обработали $5 * 3$ в Q_5 , $5 * 5$ в Q_3 :

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7 * 3, 7 * 5, 7 * 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 7 * 3, 7 * 5$ — пусть значит, мы уже обработали $7 * 3$ в Q_7 , $7 * 5$ в Q_5 :

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7 * 3, 7 * 5, 7 * 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

удален из $\pi_1 = 7 * 3, 7 * 5, 7 * 7$ — пусть значит, мы уже обработали $7 * 3$ в Q_7 , $7 * 5$ в Q_5 , $7 * 7$ в Q_3 :

$Q_3 = 3 * 3$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_5 = 5 * 3, 5 * 5$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$
$Q_7 = 7 * 3, 7 * 5, 7 * 7$	$((\langle \text{step} \rangle \rightarrow 72) \wedge \text{book}) \vee \text{book}$

8. Объектно-ориентированное проектирование

- 8.1. Разработайте структуры данных для универсальной колоды карт. Объясните, как разделить структуры данных на подклассы, чтобы реализовать игру в блэк-джек.

Решение

Прежде всего, нам нужно определиться, что такое «универсальная колода карт». «Универсальной колодой» могут назвать как стандартную колоду карт для игры в покер, так и набор бейсбольных карточек. Очень важно уточнить у интервьюера, что значит «универсальная».

Давайте предположим, что интервьюер прояснил, что мы имеем дело со стандартной колодой из 52 карт, которая используется при игре в блэк-джек или покер. Если это так, то наше 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  public int suitValue() { ... }
9  public class Deck <T extends Card> {
10     private ArrayList<T> cards; // все карты, задействованы или нет
11     private int dealtIndex = 0; // помечаем незадействованные 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     /* от двойки до десятки - числа от 2 до 10:
27      * 11 - валет, 12 - дама, 13 - король и 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
38     public Suit suit() { return suit; }
39
40     /* Проверяет, может ли карта быть передана кому-то */
41     public boolean isAvailable() { return available; }
42     public void markUnavailable() { available = false; }
43     public void markAvailable() { available = true; }
44
45 }
46
47 public class Hand <T extends Card> {
48     protected ArrayList<T> cards = new ArrayList<T>();
49
50     public int score() {
51         int score = 0;
52         for (T card : cards) {
53             score += card.value();
54         }
55         return score;
56     }
57
58     public void addCard(T card) {
59         cards.add(card);
60     }
61 }
```

В данном коде мы описали колоду (`Deck`) с некоторыми обобщениями, но ограничили типом `Card`. Мы реализовали `Card` в виде абстрактного класса, поэтому методы вроде `value()` не имеют особого смысла без привязки к конкретной игре (вы можете возразить, что, согласно правилам игры в покер, они должны быть реализованы в любом случае).

Предположим, что мы моделируем блэк-джек, поэтому нам нужно знать значения карт. Значения картинок — 10, туза — 1 или 11 (чаще всего эту задачу выполняет класс `Hand`).

```

1  public class BlackJackHand extends Hand<BlackJackCard> {
2      /* в блэк-джек есть много разных правил подсчета очков,
3      * поскольку тузы могут оцениваться по-разному. Возвращаем наибольший возможный
4      * счет, не превышающий 21, которым была закончена игра */
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  /* возвращаем список всех возможных сумм очков, которые могут находиться
20  в этой руке
21  * (туз оценивается и как 1, и как 11 */
22  private ArrayList<Integer> possibleScores() { ... }
23
24  public boolean busted() { return score() > 21; }
25  public boolean is21() { return score() == 21; }
26  public boolean isBlackJack() { ... }
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 class Cell {
47   public boolean isAce() {
48     return faceValue == 1;
49   }
50
51   public boolean isFaceCard() {
52     return faceValue >= 11 && faceValue <= 13;
53   }

```

Это один из способов обработки тузов. С другой стороны, мы могли бы создать класс типа Ace, который расширит BlackJackCards.

Полную версию этого кода можно скачать с сайта автора книги.

- 8.2. Предположим, что существует Call-центр 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, звонок перенаправляется директору. Разработайте классы и структуры данных для этой задачи. Реализуйте метод `dispatchCall()`, который перенаправляет звонок первому свободному сотруднику.

Решение

У всех служащих есть свои обязанности, которые должны выполняться, поэтому специфические функции являются профильными и должны храниться в пределах соответствующего класса.

Существует несколько общих полей, например адрес, имя, должность и возраст. Всю эту информацию имеет смысл хранить в одном классе, который может быть расширен или унаследован другими классами.

Кроме того, нам понадобится класс `CallHandler`, который перенаправляет вызов конкретному человеку.

Объектно-ориентированное проектирование предоставляет множество способов создания объектов. Обсудите различные варианты решения с интервьюером. Чаще всего от вас требуется создать гибкий и удобный в обслуживании код.

Давайте остановимся на каждом из классов поподробнее.

Класс `CallHandler` реализован как singleton-класс. Он представляет тело программы и управляет перенаправлением звонков.

```

1 public class CallHandler {
2     private static CallHandler instance;
3
4     /* 3 уровня служащих: операторы, менеджеры, директора. */
5     private final int LEVELS = 3;
6
7     /* инициализация: 10 операторов, 4 менеджера и 2 директора. */
8     private final int NUM_RESPONDENTS = 10;
9     private final int NUM_MANAGERS = 4;
10    private final int NUM_DIRECTORS = 2;
11
12    /* Список служащих, по уровням.
13     * employeeLevels[0] = операторы
14     * employeeLevels[1] = менеджеры
15     * employeeLevels[2] = директора
16     */
17    ArrayList<Employee>[] employeeLevels;
18
19    /* очереди для каждого ранга вызова */
20    Queue<Call>[] callQueues;
21
22    public CallHandler() { ... }
23
24    /* Получаем экземпляр singleton-класса. */
25    public static CallHandler getInstance() {

```

```

26     if (instance == null) instance = new CallHandler();
27     return instance;
28   }
29
30   /* Получаем первого доступного служащего, который может обработать звонок */
31   public Employee getHandlerForCall(Call call) { ... }
32
33   /* Перенаправляем звонок свободному служащему или сохраняем его в очереди,
34   * если нет свободных сотрудников. */
35   public void dispatchCall(Caller caller) {
36     Call call = new Call(caller);
37     dispatchCall(call);
38   }
39
40   /* Перенаправляем звонок свободному служащему или сохраняем его в очереди,
41   * если нет свободных сотрудников. */
42   public void dispatchCall(Call call) {
43     /* Пытаемся перенаправить звонок сотруднику с минимальным рангом */
44     Employee emp = getHandlerForCall(call);
45     if (emp != null) {
46       emp.receiveCall(call);
47       call.setHandler(emp);
48     } else {
49       /* Помещаем звонок в соответствующую рангу
50       * очередь. */
51       call.reply("Please wait for free employee to reply");
52       callQueues[call.getRank().getValue()].add(call);
53     }
54   }
55
56   /* Сотрудник освободился. Ищем звонок, который может обработать
57   * сотрудника. Возвращаем true, если звонок был присвоен, false – в противном случае. */
58   public boolean assignCall(Employee emp) { ... }
59 }
```

Класс `Call` представляет собой звонок от абонента. У звонка — минимальный ранг, это означает, что он будет перенаправлен первому свободному сотруднику:

```

1  public class Call {
2    /* Минимальный ранг сотрудника, который может обработать звонок */
3    private Rank rank;
4
5    /* Человек, который звонит. */
6    private Caller caller;
7
8    /* Сотрудник, обрабатывающий звонок. */
9    private Employee handler;
10
11   public Call(Caller c) {
12     rank = Rank.Responder;
13     caller = c;
14   }
}
```

```

15
16     /* Выбор сотрудника, который будет обрабатывать звонок */
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 реализован как абстрактный класс, поэтому нет смысла создавать экземпляр `Employee`.

```

1 abstract class Employee {
2     private Call currentCall = null;
3     protected Rank rank;
4
5     public Employee() { }
6
7     /* Запускаем разговор */
8     public void receiveCall(Call call) { ... }
9
10    /* Проблема разрешена, завершаем звонок*/
11    public void callCompleted() { ... }
12
13    /* Проблема не решена, перенаправляем
14     * звонок новому сотруднику */
15    public void escalateAndReassign() { ... }
16 }
17
18    /* Назначаем новый звонок сотруднику, если сотрудник свободен */
19    public boolean assignNewCall() { ... }
20
21    /* Возвращает true, если сотрудник свободен */
22    public boolean isFree() { return currentCall == null; }
23
24    public Rank getRank() { return rank; }
25 }

```

Классы `Respondent`, `Director` и `Manager` — просто расширение класса `Employee`.

```

1 class Director extends Employee {
2     public Director() {
3         rank = Rank.Director;
4     }
5 }
6
7 class Manager extends Employee {
8     public Manager() {
9         rank = Rank.Manager;
10    }
11 }

```

```
12
13 class Respondent extends Employee {
14     public Respondent() {
15         rank = Rank.Responder;
16     }
17 }
```

Это один из вариантов решения задачи, и таких вариантов множество.

Решение подобных задач требует написания объемного кода. Здесь приведен даже более развернутый код, чем понадобится на собеседовании. Вы можете опустить некоторые детали, поскольку на них не хватит времени.

8.3. Разработайте музыкальный автомат, используя принципы ООП.

Решение

Приступая к решению любых задач объектно-ориентированного проектирования, прежде всего выясните детали. 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?

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*).

Теперь давайте остановимся и подумаем над возможными функциями автомата:

- Создание плейлиста.
- Выбор диска.
- Выбор композиции.
- Постановка композиции в очередь.
- Выбор следующей композиции из плейлиста.

Пользователь может:

- Добавлять композицию.
- Удалять.
- Просматривать информацию о стоимости.

Каждый из компонентов системы транслируется в объект, а каждое действие преобразуется в метод. Давайте 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,
8         Set<CD> cdCollection, SongSelector ts) {
9             ...
10        }
11    abstract class Selector {
12        public Song getCurrentSong() {
13            return ts.getCurrentSong();
14        }
15        public void setUser(User u) {
16            this.user = u;
17        }
18    }
19 }
```

Подобно настоящему CD-проигрывателю, класс `CDPlayer` может читать 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 композиции */
11    public void playSong(Song s) { ... }
12
13    /* Методы для получения (get) и установки (set) плейлиста */
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` оперирует текущей и следующей композициями. На самом деле этот класс — оболочка для очереди и некоторых дополнительных методов.

```

1 public class Playlist {
2     private Song song;
3     private Queue<Song> queue;
4     public Playlist(Song song, Queue<Song> queue) {
```

```

5     ... вспомогательные методы и конструкторы ...
6 }
7     public Song getNextSongToPlay() {
8         return queue.peek();
9     }
10    public void queueUpSong(Song s) {
11        queue.add(s);
12    }
13 }

```

Классы CD, Song и User предельно просты. Они состоят из переменных экземпляра и методов для установки и получения параметров автомата.

```

1 public class CD {
2     /* данные для id, artist, songs и т.д. */
3 }
4
5 public class Song {
6     /* данные для id, CD (может быть null), title, length и т.д. */
7 }
8
9 public class User {
10    private String name;
11    public String getName() { return name; }
12    public void setName(String name) { this.name = name; }
13    public long getID() { return ID; }
14    public void setID(long ID) { ID = iD; }
15    private long ID;
16    public User(String name, long iD) { ... }
17    public User getUser() { return this; }
18    public static User addUser(String name, long iD) { ... }
19 }

```

Данная реализация — не единственно правильная. Ответы интервьюера на ваши вопросы помогут сформировать другой набор классов.

8.4. Разработайте паркинг, используя принципы ООП.

Решение

Формулировку этого задания нельзя назвать конкретной. Прежде чем приступить к решению, нужно расспросить интервьюера про нюансы задачи — какие транспортные средства используются на автостоянке, сколько должно быть 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 5 больших мест, которые должны быть расположены последовательно в одном ряду. Автобус нельзя парковать на мотоциклетных и компактных парковочных местах.

Давайте создадим абстрактный класс `Vehicle`, который наследуют классы `Car`, `Bus` и `Motorcycle`. Для обработки 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 =
5          new ArrayList<ParkingSpot>();
6      protected String licensePlate;
7      protected int spotsNeeded;
8      protected VehicleSize size;
9
10     public int getSpotsNeeded() { return spotsNeeded; }
11     public VehicleSize getSize() { return size; }
12
13     /* Паркуем транспортное средство на парковочном месте */
14     public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
15
16     /* "Удаляем" машину, уведомляем о доступном месте */
17     public void clearSpots() { ... }
18
19     /* Проверяем, достаточно ли места для парковки транспортного средства (ТС).
20      * Этот метод только сравнивает размер. Он не проверяет, достаточно ли мест
21      * для парковки ТС. */
22     public abstract boolean canFitInSpot(ParkingSpot spot);
23 }
24
25 public class Bus extends Vehicle {
26     public Bus() {
27         spotsNeeded = 5;
28         size = VehicleSize.Large;
29     }
30
31     /* Проверяем, является ли место большим. Не проверяем количество мест */
32     public boolean canFitInSpot(ParkingSpot spot) { ... }
33 }
34
35 public class Car extends Vehicle {
36     public Car() {
37         spotsNeeded = 1;
38         size = VehicleSize.Compact;
39     }
40
41     /* Проверяем, является место компактным или большим */
42     public boolean canFitInSpot(ParkingSpot spot) { ... }
43 }
```

```

44 был реализован для оценки, что в парковке есть место для машины и т.д.
45 public class Motorcycle extends Vehicle {
46     public Motorcycle() {
47         spotsNeeded = 1;
48         size = VehicleSize.Motorcycle;
49     }
50
51     public boolean canFitInSpot(ParkingSpot spot) { ... }
52 }

```

Класс `ParkingLot` — это оболочка для массива `Levels`. Благодаря такой реализации можно выделить логику, связанную с фактическим нахождением свободных мест и парковкой автомобилей. Альтернативный подход — хранение информации о парковочных местах в двойном массиве (или хэш-таблице, которая ставит соответствие между номером уровня и списком мест). Но первый подход понятнее благодаря отделению `ParkingLot` от `Level`.

```

1  public class ParkingLot {
2      private Level[] levels;
3      private final int NUM_LEVELS = 5;
4
5      public ParkingLot() { ... }
6
7      /* Паркуем автомобиль на парковочном месте (или нескольких местах).
8       * Возвращаем false в случае неудачи. */
9      public boolean parkVehicle(Vehicle vehicle) { ... }
10 }
11
12 /* Представляет уровень парковки */
13 public class Level {
14     private int floor;
15     private ParkingSpot[] spots;
16     private int availableSpots = 0; // число свободных мест
17     private static final int SPOTS_PER_ROW = 10;
18
19     public Level(int flr, int numberSpots) { ... }
20
21     public int availableSpots() { return availableSpots; }
22
23     /* Находит место для парковки транспортного средства. Возвращает false в случае
24      неудачи. */
25     public boolean parkVehicle(Vehicle vehicle) { ... }
26
27     /* Паркует ТС начиная с места spotNumber и до
28      * vehicle.spotsNeeded. */
29     private boolean parkStartingAtSpot(int num, Vehicle v) { ... }
30
31     /* Находит место для парковки ТС. Возвращает индекс места или -1
32      * в случае неудачи. */
33     private int findAvailableSpots(Vehicle vehicle) { ... }

```

продолжение ⤵

```

34     /* Когда машина освобождает место, нужно
35      * увеличить значение availableSpots */
36     public void spotFreed() { availableSpots++; }
37 }

```

В `ParkingSpot` есть переменная, описывающая размер парковочного места. Можно реализовать дополнительные классы `LargeSpot`, `CompactSpot` и `MotorcycleSpot`, унаследованные от `ParkingSpot`, но это будет избыточно. Поведение парковочного места неизменно, 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     /* Проверяет, достаточно ли места для парковки и свободно ли парковочное место */
13     public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15     /* Паркует ТС на это место */
16     public boolean park(Vehicle v) { ... }
17
18     public int getRow() { return row; }
19     public int getSpotNumber() { return spotNumber; }
20
21     /* Удаляет ТС, уведомляет о свободном
22      * месте */
23     public void removeVehicle() { ... }
24 }

```

Полная версия этого кода, в том числе исполнимый код, предоставлена в загружаемом вложении.

8.5. 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:

- Ведение списка подписчиков.
- Поиск книг в базе.
- Чтение.
- В каждый момент времени только один пользователь может быть активен.
- Пользователь может читать только одну книгу.

Чтобы реализовать эти операции, нам понадобятся функции `get`, `set`, `update` и т. д. Объектами нашей системы будут `User`, `Book` и `Library`.

Класс `OnlineReaderSystem` является сутью программы. Можно реализовать класс так, что он будет хранить информацию обо всех книгах, управлять пользователями, обновлять экран, но в итоге класс окажется слишком большим. Вместо этого давайте разделим 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 отображением будут заниматься разные классы:

```

1 public class Library {
2     private Hashtable<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}
```

продолжение ↗

```

12
13     public boolean remove(Book b) { return remove(b.getID()); }
14
15     public boolean remove(int id) {
16         if (!books.containsKey(id)) {
17             return false;
18         }
19         books.remove(id);
20         return true;
21     }
22
23     public Book find(int id) {
24         return books.get(id);
25     }
26
27     public class UserManager {
28         private Hashtable<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         public void setActiveBook() {
39             public int activeUser() { return -1; }
40             public boolean remove(User u) {
41                 return remove(u.getID());
42             }
43             public boolean remove(int id) {
44                 if (!users.containsKey(id)) {
45                     return false;
46                 }
47                 users.remove(id);
48                 return true;
49             }
50
51             public User find(int id) {
52                 return users.get(id);
53             }
54         }
55
56         public class Display {
57             private Book activeBook;
58             private User activeUser;
59             private int pageNumber = 0;
60
61             public void displayUser(User user) {
62                 activeUser = user;

```

```

63     refreshUsername();
64 }
65
66 public void displayBook(Book book) {
67     pageNumber = 0;
68     activeBook = book;
69
70     refreshTitle();
71     refreshDetails();
72     refreshPage();
73 }
74
75 public void turnPageForward() {
76     pageNumber++;
77     refreshPage();
78 }
79
80 public void turnPageBackward() {
81     pageNumber--;
82     refreshPage();
83 }
84
85 public void refreshUsername() { /* обновляем имя пользователя */ }
86 public void refreshTitle() { /* обновляем заголовок */ }
87 public void refreshDetails() { /* обновляем экран с подробностями */ }
88 public void refreshPage() { /* обновляем экран страницы */ }
89 }

```

Классы User и Book используются для хранения 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() { }

```

продолжение ↗

```

22
23     public User(int id, String details, int accountType) {
24         userId = id;
25         this.details = details;
26         this.accountType = accountType;
27     }
28
29     /* Методы получения и установки параметров */
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 }

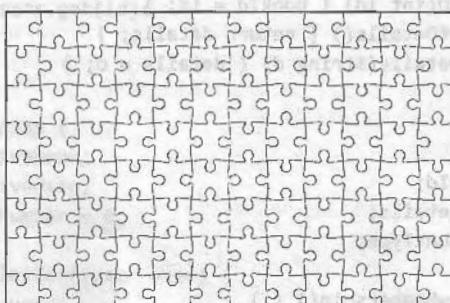
```

Решение вынести управление пользователями, библиотекой и экраном в отдельные классы из класса `onlinereadersystem` является довольно интересным. При разработке небольшой системы это усложнит код, но если система будет развиваться, то лучше вынести дополнительный функционал в разные классы, иначе `onlinereadersystem` станет огромной.

- 8.6.** Запрограммируйте игру в пазл. Разработайте структуры данных и объясните алгоритм, позволяющий решить задачу. Вы можете предположить, что существует метод `fitsWith`, который возвращает значение `true`, если два переданных кусочка пазла должны располагаться рядом.

Решение

Мы предполагаем, что имеем дело с традиционным, простым пазлом, который, подобно таблице, делится на колонки и строки. Каждый кусочек пазла привязан к строке и колонке и имеет четыре стороны. Каждая сторона может относиться к одному из трех типов: вогнутая, выпуклая и плоская. Угловой элемент, например, имеет две плоских стороны и две стороны другого типа, которые могут быть как вогнутыми, так и выпуклыми.



Когда мы собираем пазл (вручную или алгоритмически), нужно где-то хранить расположение каждого фрагмента. Позиция может быть абсолютной или относительной:

- Абсолютная позиция — «Фрагмент находится на позиции (12, 23)». Абсолютная позиция описывается классом `Piece`.
- Относительная позиция — «Я не знаю, где расположен этот кусочек, но я знаю, что он находится рядом с другим». Относительная позиция описывается классом `Edge`.

Для построения решения мы будем использовать только относительную позицию, соединяя края каждого фрагмента с соседними краями.

Объектно-ориентированный проект будет иметь следующий вид:

```

1 class Edge {
2     enum Type { inner, outer, flat }
3     Piece parent;
4     Type type;
5     int index; // индекс Piece.edges
6     Edge attached_to; // относительная позиция
7
8     /* См. раздел Алгоритм. Возвращает true, если две части
9      * должны соприкасаться друг с другом. */
10    boolean fitsWith(Edge edge) { ... };
11 }
12
13 class Piece {
14     Edge[] edges;
15     boolean isCorner() { ... }
16 }
17
18 class Puzzle {
19     Piece[] pieces; /* оставшиеся части. */
20     Piece[][] solution;
21
22     /* см. раздел Алгоритм. */
23     Edge[] inners, outers, flats;
24     Piece[] corners;
25
26     /* см. раздел Алгоритм. */
27     void sort() { ... }
28     void solve() { ... }
29 }
```

Алгоритм сборки пазла

Давайте разработаем алгоритм, используя смесь псевдокода с реальными фрагментами программы.

Подобно ребенку, который мог бы решать эту задачу, давайте начнем с самых легких частей: углы и края. Мы можем проанализировать все фрагменты, чтобы найти края. Имеет смысл сгруппировать все фрагменты по типу сторон.

```

1 void sort() {
2     for each Piece p in pieces {
3         если у p две плоские границы, то добавить p в corners
4         для каждой границы в p.edges {
5             если сторона вогнутая, добавить к inners
6             если сторона выпуклая, добавить кouters
7         }
8     }
9 }
```

Давайте решать задачу последовательно, линия за линией, сопоставляя фрагменты. Метод `solve`, приведенный ниже, начинает работу с произвольного угла. Затем он находит фрагмент краевого элемента и пытается сопоставить ее с открытой частью. Когда совпадение найдено, он:

1. Присоединяет сторону.
2. Удаляет сторону из списка открытых.
3. Находит следующую открытую сторону.

Следующая открытая сторона должна примыкать непосредственно к текущей, если это возможно. Если не получается, то можно взять любую другую сторону. Таким образом, мы сможем собрать пазл по спирали, двигаясь от краев к центру.

Алгоритм двигается по прямой, пока это возможно. Когда мы достигаем конца первой стороны, алгоритм делает поворот на 90 градусов и начинает работать с другой свободной границей. Двигаясь таким образом, алгоритм продолжает делать повороты на 90 градусов, пока вся внешняя граница не будет пройдена. Алгоритм повторяет свою работу вновь и вновь, пока все части не будут на месте.

Ниже приведен алгоритм, записанный на Java-подобном псевдокоде:

```

1 public void solve() {
2     /* Выбираем любой угол для старта */
3     Edge currentEdge = getExposedEdge(corner[0]);
4
5     /* Цикл будет двигаться по спирали, пока
6      * пазл не будет собран. */
7     while (currentEdge != null) {
8         /* Сопоставление сторон. Внутренние - внешние и т.д. */
9         Edge[] opposites = currentEdge.type == inner ?
10             outers : inners;
11
12         for each Edge fittingEdge in opposites {
13             if (currentEdge.fitsWith(fittingEdge)) {
14                 attachEdges(currentEdge, fittingEdge); //присоединяем границы
15                 removeFromList(currentEdge);
16                 removeFromList(fittingEdge);
17
18                 /* переходим к следующей границе */
19                 currentEdge = nextExposedEdge(fittingEdge);
20                 break; // Прерываем внутренний и продолжаем внешний цикл
21             }
22         }
23 }
```

```
24
25 public void removeFromList(Edge edge) {
26     if (edge.type == flat) return;
27     Edge[] array = currentEdge.type == inner ? inners : outers;
28     array.remove(edge);
29 }
30
31 /* Возвращаем противоположную границу, если возможно. Иначе возвращаем
32 * любую открытую границу. */
33 public Edge nextExposedEdge(Edge edge) {
34     int next_index = (edge.index + 2) % 4; // Противоположная граница
35     Edge next_edge = edge.parent.edges[next_index];
36     if isExposed(next_edge) {
37         return next_edge;
38     }
39     return getExposedEdge(edge.parent);
40 }
41
42 public Edge attachEdges(Edge e1, Edge e2) {
43     e1.attached_to = e2;
44     e2.attached_to = e1;
45 }
46
47 public Edge isExposed(Edge e1) {
48     return edge.type != flat && edge.attached_to == null;
49 }
50
51 public Edge getExposedEdge(Piece p) {
52     for each Edge edge in p.edges {
53         if (isExposed(edge)) {
54             return edge;
55         }
56     }
57     return null;
58 }
```

Мы храним *inners* и *outers* в массиве *Edge*. На самом деле это удачное решение, если придется часто добавлять и удалять элементы. Если бы речь шла о настоящей программе, то эти переменные нужно было бы представить как связные списки. Написание полного кода для такой задачи выходит за рамки собеседования. Скорее всего, вас попросят написать псевдокод.

8.7. Как вы будете разрабатывать чат-сервер? Предоставьте информацию о компонентах бэкэнда, классах и методах. Перечислите самые трудные задачи, которые необходимо решить.

Решение

Разработка чат-сервера — сложный проект, и вряд ли вам удастся реализовать его в рамках собеседования. В конце концов, команды из многих людей проводят ме-

сяцы и годы, пытаясь создать хороший чат-сервер. Вы, как кандидат, должны сфокусироваться на задаче в целом, но так, чтобы уложиться в отведенное время.

В нашем случае мы займемся основными аспектами работы с участниками чата и организацией общения: добавление пользователя, создание беседы, обновление статуса и т. д. Из-за дефицита времени мы не будем углубляться в сетевые аспекты задачи и не будем рассматривать вопросы передачи данных между клиентами.

Предположим, что «дружба» может быть только взаимной: если вы дружите со мной то и я дружи с вами. Наша чат-система будет поддерживать как групповое так и персональное общение. Давайте не будем рассматривать голосовые и видеофункции а также передачу файлов.

Задачи, которые необходимо решить

Этот вопрос следует обсудить с интервьюером. Задачи могут быть такие:

- Вход и выход из чата.
- Добавление запроса (отправка, прием и отклонение).
- Обновление информации о статусе.
- Создание персональных и групповых чатов.
- Добавление новых сообщений.

Это далеко не полный список. Если в вашем распоряжении будет достаточно времени, можно добавить дополнительные действия.

Требования к чат-серверу

Нам необходима концепция пользователей, статуса добавления запроса, статуса подключения и сообщений.

Базовые компоненты системы

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 как место для 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 систему.

Для обмена данными между клиентами и серверами подойдет XML. Хотя это не самый плотный формат (вы обязательно должны сказать об этом интервьюеру), но он наиболее удобен для восприятия как человеком, так и компьютером. Используя XML, вы обеспечите простую 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, придется перераспределять некоторые данные между машинами, чтобы сократить время поиска. Нужно избегать узких мест. Например, если аутентификацией пользователей занимается только одна машина, то при выходе ее из строя мы отключим миллионы пользователей.

Основные объекты и методы

Ключевыми объектами нашей системы являются пользователи, беседы и сообщения о статусах. Все это можно реализовать с помощью класса `UserManagement`. Если бы мы рассматривали сетевые аспекты задачи или конкретные компоненты, то нам пришлось бы создавать дополнительные объекты.

```

1  /* UserManager обслуживает основные операции пользователей */
2  public class UserManager {
3      private static UserManager instance;
4      /* преобразует id в пользователя */
5      private HashMap<Integer, User> usersById;
6
7      /* ставит в соответствие учетную запись и пользователя */
8      private HashMap<String, User> usersByAccountName;
9
10     /* преобразует id в онлайн-пользователя */
11     private HashMap<Integer, User> onlineUsers;
12
13     public static UserManager getInstance() {
14         if (instance == null) instance = new UserManager();
15         return instance;
16     }
17
18     public void addUser(User fromUser, String toAccountName) { ... }
19     public void approveAddRequest(AddRequest req) { ... }
20     public void rejectAddRequest(AddRequest req) { ... }
21     public void userSignedOn(String accountName) { ... }
22     public void userSignedOff(String accountName) { ... }
23 }
```

Метод `receivedAddRequest` в классе `User` уведомляет `User B`, что `User A` хочет добавить его в список контактов. `User B` может согласиться на добавление или отказать (с помощью `UserManager.approveAddRequest` или `rejectAddRequest`), а `UserManager` позаботится о добавлении пользователей в списки контактов друг друга.

Для добавления `AddRequest` в список запросов пользователя `UserManager` вызывает метод `sentAddRequest` из класса `User`. Итак, картина следующая:

1. `User A` нажимает кнопку «добавить пользователя», и запрос отправляется на сервер.
2. `User A` вызывает `requestAddUser(User B)`.
3. Данный метод вызывает `UserManager.addUser()`.
4. `UserManager` вызывает оба метода, `User A.sentAddRequest` и `User B.receivedAddRequest`.

Это один из 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      /* преобразует id другого участника в чат */
6      private HashMap<Integer, PrivateChat> privateChats;
7 }
```

продолжение ↗

```

8     /* преобразует id группы чата в название группы чата */
9     private ArrayList<GroupChat> groupChats;
10
11    /* преобразует id другого человека в запрос на добавление */
12    private HashMap<Integer, AddRequest> receivedAddRequests;
13
14    /* преобразует id другого человека в запрос на добавление */
15    private HashMap<Integer, AddRequest> sentAddRequests;
16
17    /* преобразует id пользователя в запрос на добавление */
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 класс, поэтому его потомки (`GroupChat` или `PrivateChat`) наделены 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 }
```

```

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 простой функциональностью. Основное их назначение — группировать данные, которые используют другие классы.

```

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 }
```

Полную версию этого кода можно скачать с сайта автора книги. (Загружаемый код содержит более подробное описание этих методов, включая реализацию методов, описанных ранее.)

Какие более сложные (или интересные) задачи нужно решать?

Приведенные далее вопросы стоит обсудить с интервьюером.

Вопрос 1. Как узнать, кто находится онлайн, я имею в виду на самом деле онлайн?

Пока пользователь не выйдет из системы, мы не можем быть уверены, что он присутствует онлайн. Соединение может быть разорвано. Чтобы убедиться, что пользователь

вышел на самом деле, нужно регулярно пинговать клиента пользователя, чтобы удостовериться, что пользователь еще в чате.

Вопрос 2. Что мы будем делать со сбоями?

Часть информации должна храниться в оперативной памяти, а часть — в базе данных. Что случится, если произойдет сбой синхронизации? Какая информация должна считаться правильной?

Вопрос 3. Что делать с масштабируемостью сервера?

Мы проектировали чат-сервер, не заботясь о масштабируемости. Но в реальной ситуации придется решать и эту задачу. Нам нужно разделить данные на несколько серверов и реализовать механизм синхронизации данных.

Вопрос 4. Как защититься от DOS-атак?

Клиенты могут передавать нам данные, а что если нас попытаются «задосить»? Как предотвратить атаку?

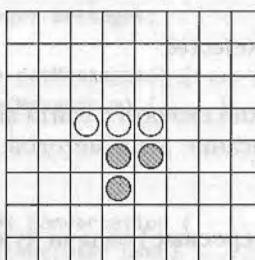
- 8.8. Правила игры «Реверси» следующие. Каждая фишка в игре с одной стороны белая, а с другой — черная. Когда ряд фишек оказывается ограничен фишками противника (слева и справа или сверху и снизу), его цвет меняется на противоположный. Цель — захватить по крайней мере одну из фишек противника. Игра заканчивается, когда у игрока не остается ходов. Побеждает тот, у которого больше фишек на поле. Реализуйте ООП-модель для этой игры.**

Решение

Давайте начнем с примера. Предположим, что правила «Реверси» следующие:

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:



Ключевые объекты — сама игра, доска, фишки (черные или белые) и, конечно же, игроки. Как представить все это с помощью объектно-ориентированного проектирования?

Должны ли BlackPiece и WhitePiece быть классами?

При первом взгляде на задачу кажется, что нам понадобятся два класса, `BlackPiece` и `WhitePiece`, которые наследуются от абстрактного класса `Piece`. Однако это не самая лучшая идея. Каждая фишка может менять цвет, следовательно, нам придется постоянно уничтожать и создавать объект «указанного цвета», что не очень практично. Гораздо лучше 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 нам логически разделить функциональные задачи между доской (логика перемещения фишек) и игрой (время игры, игровой поток и т. д.). В такой системе есть небольшой недостаток — мы добавляем дополнительные уровни в нашу программу. Мы разделим объекты `Game` и `Board`, но на собеседовании вам следует обсудить этот момент с интервьюером.

Где будет храниться результат?

Нам необходимо хранить количество черных и белых фишек. Но кто будет отвечать за эту информацию? `Game`, `Board` или `Piece` (в статических методах). В нашем случае мы будем хранить эту информацию в `Board`. Она обновляется объектами `Piece` или `Board` при вызове методов `colorChanged` и `colorAdded`.

Должен ли Game быть singleton-классом?

Реализация `Game` в виде singleton-класса имеет определенное преимущество — легко можно вызывать метод из `Game`, не передавая ссылку на объект `Game`.

Но это означает, что у нас будет только один экземпляр `Game`. Можем ли мы пойти на это? Лучше обсудить этот нюанс с интервьюером.

Одна из возможных реализаций «Реверси» представлена ниже:

```

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];
}

```

продолжение

```

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        /* инициализируем центральные черные и белые фишки */
12    }
13
14    /* Пытаемся поместить фишку цвета color на позицию (row, column).
15     * Возвращает true, если успешно. */
16    public boolean placeColor(int row, int column, Color color) {
17        ...
18    }
19
20    /* переворачиваем фишки, начиная с (row, column) и продолжая в
21     * направлении d. */
22    private int flipSection(int row, int column, Color color,
23    Direction d) { ... }
24
25    public int getScoreForColor(Color c) {
26        if (c == Color.Black) return blackCount;
27        else return whiteCount;
28    }
29
30    /* Обновляем доску дополнительными фишками с цветом NewColor
31     * Уменьшаем количество фишек противоположного цвета. */
32    public void updateScore(Color newColor, int newPieces) { ... }
33 }
```

Как показано выше, мы реализовали черные и белые фишкы с помощью класса `Piece`, у которого есть переменная `Color`, соответствующая цвету фишкы — `Black` или `White`.

```

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 мало информации. В нем нет информации о счете, но у него есть метод, позволяющий получить счет конкретного игрока, — `getScore()`. Этот метод обращается к `GameManager`, чтобы получить соответствующее значение.

```

12 public class Player {
13     private Color color;
14     public Player(Color c) { color = c; }
15
16     public int getScore() { ... }
17
18     public boolean playPiece(int r, int c) {
19         return Game.getInstance().getBoard().placeColor(r, c, color);
20     }
21
22     public Color getColor() { return color; }
23 }
```

Полностью функциональную (автоматическую) версию этого кода можно скачать с сайта.

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 виде `singleton`-класса или нет, ему важно, чтобы вы учли все нюансы.

8.9. Объясните, какие структуры данных и алгоритмы необходимо использовать 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 задачам нижнего уровня!

Не паникуйте! Если правильно выбрать компоненты файловой системы, то можно превратить эту задачу в задачу ООП.

Файловая система в самом простом случае состоит из файлов (`Files`) и папок (`Directories`). Каждая из папок в свою очередь может содержать множество файлов

и папок. У файла и папки 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     /* Методы установки и получения параметров. */
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++; // Directory counts as a file
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 }

```

Можно воспользоваться и другим вариантом — реализовать класс `Directory` так, чтобы в нем можно было хранить отдельно списки для файлов и папок. Это сделает метод `numberOfFiles()` более понятным, поскольку не нужно будет использовать оператор `instanceof`, но усложнит задачу сортировки файлов и папок по имени или дате.

8.10. Спроектируйте и реализуйте хэш-таблицу, использующую связные списки для обработки коллизий.

Решение

Предположим, что мы реализуем хэш-таблицу, которая имеет вид `Hash<K, V>` и представляет собой соответствие объектов типа `K` объектам типа `V`.

Наша структура данных может иметь, например, следующий вид:

```
1 public class Hash<K, V> {
2     LinkedList<V>[] items;
3     public void put(K key, V value) { ... }
4     public V get(K key) { ... }
5 }
```

`items` — это массив связных списков, а `items[i]` — связный список всех объектов с ключами, которым сопоставляется индекс `i` (все коллизии с индексом `i`).

Все будет работать, пока мы не задумаемся о коллизиях.

Допустим, что у нас есть простая хэш-функция, которая использует длину строки:

```
1 public int hashCodeOfKey(K key) {
2     return key.toString().length() % items.length;
3 }
```

Ключи `jim` и `bob` будут отражены в индекс массива, даже если это разные ключи (ведь длина строки одинакова). Нам нужно произвести поиск по связному списку, чтобы найти правильные объекты, соответствующие ключам. Но как это осуществить? Ведь мы храним в связном списке значение, а не ключ.

Один из способов — создание другого объекта (`Cell`), в котором будет храниться пара «ключ—значение». В этой реализации наш связный список будет иметь тип `Cell`.

Приведенный далее код использует эту реализацию:

```
1 public class Hash<K, V> {
2     private final int MAX_SIZE = 10;
3     LinkedList<Cell<K, V>>[] items;
4
5     public Hash() {
6         items = (LinkedList<Cell<K, V>>[]) new LinkedList[MAX_SIZE];
7     }
8
9     /* Действительно, самый простой хэш */
10    public int hashCodeOfKey(K key) {
11        return key.toString().length() % items.length;
12    }
13
14    public void put(K key, V value) {
15        int x = hashCodeOfKey(key);
16        if (items[x] == null) {
17            items[x] = new LinkedList<Cell<K, V>>();
18        }
19        LinkedList<Cell<K, V>> collided = items[x];
20
21        /* Ищем элементы с тем же самым ключом и заменяем их, если они найдены */
22        for (Cell<K, V> c : collided) {
23            if (c.equivalent(key)) {
24                collided.remove(c);
25                break;
26            }
27        }
28    }
```

```

29
30     Cell<K, V> cell = new Cell<K, V>(key, value);
31     collided.add(cell);
32 }
33
34 public V get(K key) {
35     int x = hashCodeOfKey(key);
36     if (items[x] == null) {
37         return null;
38     }
39     LinkedList<Cell<K, V>> collided = items[x];
40     for (Cell<K, V> c : collided) {
41         if (c.equivalent(key)) {
42             return c.getValue();
43         }
44     }
45     return null;
46 }
47 }
48 }
```

Класс `Cell` содержит пары данных — значение и ключ. Это позволяет производить поиск по связному списку и находить объекты с точным значением ключа.

```

1 public class Cell<K, V> {
2     private K key;
3     private V value;
4     public Cell(K k, V v) {
5         key = k;
6         value = v;
7     }
8
9     public boolean equivalent(Cell<K, V> c) {
10         return equivalent(c.getKey());
11     }
12
13     public boolean equivalent(K k) {
14         return key.equals(k);
15     }
16
17     public K getKey() { return key; }
18     public V getValue() { return value; }
19 }
```

Еще один способ построения хэш-таблицы предусматривает бинарное дерево поиска в качестве основной структуры данных. Получение элемента потребует не более $O(1)$ времени (хотя в реальности это значение будет больше, если коллизий много), зато не нужно создавать огромный массив и хранить элементы.

9. Рекурсия и динамическое программирование

- 9.1.** Ребенок поднимается по лестнице из n ступенек, он может прыгнуть на одну, две или три ступеньки за один шаг. Реализуйте метод, рассчитывающий количество возможных вариантов прохождения ребенка по лестнице.

Решение

Давайте решим эту задачу сверху вниз. На последнем прыжке (n -я ступенька) ребенок может совершить одиничный, двойной или тройной прыжок. Таким образом, последний прыжок будет перемещением со ступеньки $n-1$ (ребенок перепрыгнул одну ступеньку), $n-2$ (две) или $n-3$ (три). Общее количество способов попадания на последний шаг является суммой способов попадания на каждый из трех вариантов последнего шага.

Простая реализация кода имеет вид:

```
1 public 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) +
8             countWays(n - 3);
9     }
10 }
```

Подобно задаче Фибоначчи, время выполнения этого алгоритма растет экспоненциально $O(N^3)$, поскольку каждый вызов разветвляется на три. Это означает, что `countWays` многократно вызывается для одного и того же значения. Мы можем исправить этот недостаток с помощью динамического программирования.

```
11 public static int countWaysDP(int n, int[] map) {
12     if (n < 0) {
13         return 0;
14     } else if (n == 0) {
15         return 1;
16     } else if (map[n] > -1) {
17         return map[n];
18     } else {
19         map[n] = countWaysDP(n - 1, map) +
20             countWaysDP(n - 2, map) +
21             countWaysDP(n - 3, map);
22         return map[n];
23     }
24 }
```

Будете вы использовать динамическое программирование или нет, в любом случае обратите внимание, что вы быстро перестанете помещаться в диапазон `int`. Переполнение возникнет, как только вы доберетесь до $n = 37$. Использование типа `long` отсрочит проблему, но не решит ее.

- 9.2.** Представьте себе робота, сидящего в левом верхнем углу сетки с координатами X, Y . Робот может перемещаться в двух направлениях: вправо и вниз. Сколько существует маршрутов, проходящих от точки $(0, 0)$ до точки (X, Y) .

Дополнительно:

Предположите, что на сетке существуют области, которые робот не может пересекать. Разработайте алгоритм построения маршрута от левого верхнего до правого нижнего угла.

Решение

Нам нужно подсчитать количество вариантов прохождения дистанции с X шагов вправо и Y шагов вниз ($X + Y$ шагов).

Чтобы создать путь, мы делаем X шагов вправо так, чтобы общее количество перемещений оставалось фиксированным ($X + Y$). Таким образом, количество путей должно совпадать с количеством способов выбрать X элементов из $X + Y$, то есть биномиальным коэффициентом. Биномиальный коэффициент из n по r имеет вид:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

Для нашей задачи выражение будет следующим:

$$\binom{X+Y}{X} = \frac{(X+Y)!}{X!Y!}$$

Даже если вы не знакомы с комбинаторикой, то все равно можете найти решение этой задачи самостоятельно.

Представим путь как строку длиной $X + Y$, состоящую из X символов R и Y символов D. Мы знаем, что из $X + Y$ неповторяющихся символов мы можем составить $(X+Y)!$ строк. Но в нашем случае используется X символов R и Y символов D. Символы R могут быть расставлены в строке $X!$ способами (то же самое мы можем сделать и с символами D). Таким образом, необходимо убрать лишние строки $X!$ и $Y!$. В итоге мы получим то же самое выражение:

$$\frac{(X+Y)!}{X!Y!}$$

Дополнительно: найдите маршрут (на карте есть места, через которые не может пройти робот)

Если мы изобразим нашу карту, то единственный способ попасть в квадрат (x, y) — оказаться в одном из смежных квадратов: $(x-1, y)$ или $(x, y-1)$. Следовательно, необходимо найти путь к любому из этих квадратов $((x-1, y)$ или $(x, y-1)$).

Как это осуществить? Чтобы найти путь в квадрат $(x-1, Y)$ или $(x, Y-1)$, мы должны оказаться в одной из смежных ячеек. То есть нам необходимо найти путь к квадрату, смежному с $(x-1, Y)$ ($(x-2, Y)$ и $(x-1, Y-1)$) или $(x, Y-1)$ ($(x-1, Y-1)$ и $(x, Y-2)$). Обратите внимание: в наших рассуждениях точка $(x-1, Y-1)$ упоминается дважды, мы еще вернемся к этому факту.

Давайте попробуем найти путь от исходного квадрата, двигаясь в обратном направлении, — начинаем с последней ячейки и пытаемся найти путь к каждому смежному квадрату. Далее приведен рекурсивный код, реализующий наш алгоритм:

```

1 public boolean getPath(int x, int y, ArrayList<Point> path) {
2     Point p = new Point(x, y);
3     path.add(p);
4     if (x == 0 && y == 0) {
5         return true; // найти путь
6     }
7     boolean success = false;
8     if (x >= 1 && isFree(x - 1, y)) { // Пытаемся идти вправо
9         success = getPath(x - 1, y, path); // Свободно! Можно идти вправо
10    }
11    if (!success && y >= 1 && isFree(x, y - 1)) { // Пытаемся идти вниз
12        success = getPath(x, y - 1, path); // Свободно! Можно идти вниз
13    }
14    if (!success) {
15        path.remove(p); // Неверный путь! Прекратить движение этим маршрутом
16    }
17    return success;
18 }
```

Помните, что маршруты дублируются? Чтобы найти все пути к (x, Y) , мы находим все пути к $(x-1, Y)$ и $(x, Y-1)$. Затем мы смотрим на координаты смежных квадратов: $(x-2, Y)$, $(x-1, Y-1)$, $(x-1, Y-1)$ и $(x, Y-2)$. Квадрат $(x-1, Y-1)$ появляется дважды. Давайте будем запоминать посещенные квадраты, чтобы не тратить на них время.

Это можно сделать с помощью следующего алгоритма динамического программирования:

```

1 public boolean getPath(int x, int y, ArrayList<Point> path,
2 Hashtable<Point, Boolean> cache) {
3     Point p = new Point(x, y);
4     if (cache.containsKey(p)) { // Мы уже посещали эту ячейку
5         return cache.get(p);
6     }
7     path.add(p);
8     if (x == 0 && y == 0) {
9         return true; // найден путь
10    }
11    boolean success = false;
12    if (x >= 1 && isFree(x - 1, y)) { // Пытаемся идти вправо
13        success = getPath(x - 1, y, path, cache); // Свободно! Можно идти вправо
14    }
15    if (!success && y >= 1 && isFree(x, y - 1)) { // Пытаемся идти вниз
16        success = getPath(x, y - 1, path, cache); // Свободно! Можно идти вниз
17    }
18 }
```

```

18     if (!success) {
19         path.remove(p); // Неверный путь! Прекратить движение этим маршрутом
20     }
21     cache.put(p, success); // Вычисляем результат
22     return success;
23 }
```

Это простое изменение сделает наш код более быстрым.

- 9.3.** В массиве $A[0..n-1]$ задан «волшебный» индекс — $A[i]=i$. Учитывая, что массив отсортирован, напишите метод, чтобы найти этот «волшебный» индекс, если он существует в массиве A .

Дополнительно:

Что произойдет, если значения окажутся одинаковыми?

Решение

Можно решать такую задачу «в лоб», и в таком подходе нет ничего зазорного. Мы просто пройдемся по массиву и отыщем элемент, соответствующий условию.

```

1 public static 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 }
```

Учитывая, что в условии задачи сказано, что массив отсортирован, вероятно, мы должны использовать эту информацию.

Эта задача похожа на классическую задачу бинарного поиска. Для алгоритма больше всего подходит способ «Сопоставление с образцом».

При бинарном поиске мы находим элемент k , сравнивая его со средним элементом, x , чтобы определить, по какую сторону от x находится k — слева или справа.

Давайте попробуем определить, где может находиться «волшебный» элемент. Взгляните на массив:

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

Если взять средний элемент $A[5] = 3$, то становится ясно, что «волшебный» элемент должен находиться правее, так как $A[mid] < mid$.

Почему этот элемент не может быть слева? При движении в направлении от i к $i-1$ значение элемента должно уменьшиться не менее чем на 1 (так как массив отсортирован и не содержит одинаковых элементов). Если средний элемент меньше искового, то при движении влево, смешаясь на k индексов и (как минимум) на k значений, мы будем попадать на еще более маленькие значения.

При использовании рекурсивного решения алгоритм будет похож на бинарный поиск.

```

1 public static int magicFast(int[] array, int start, int end) {
2     if (end < start || start < 0 || end >= array.length) {
3         return -1;
4     }
5     int mid = (start + end) / 2;
6     if (array[mid] == mid) {
7         return mid;
8     } else if (array[mid] > mid) {
9         return magicFast(array, start, mid - 1);
10    } else {
11        return magicFast(array, mid + 1, end);
12    }
13 }
14
15 public static int magicFast(int[] array) {
16     return magicFast(array, 0, array.length - 1);
17 }

```

Дополнительно: что если элементы повторяются?

Если элементы массива повторяются, то наш алгоритм не будет работать. Давайте 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.

Может ли он находиться слева? Нет. Так как $A[5] = 3$, мы знаем, что $A[4]$ не может быть «волшебным» элементом. $A[4]$ должен быть равен 4, но в то же время мы знаем, что $A[4]$ должен быть меньше, чем $A[5]$.

Фактически, когда мы видим, что $A[5] = 3$, нам достаточно проанализировать только правую сторону, 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 $A[3]$ — это первый элемент, который может быть «волшебным».

В общем, нам нужно сравнить `midIndex` и `midValue`. Если они не совпадают, то следует начать поиск в левой (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`.

Представленный ниже код реализует данный алгоритм:

```

1 public static int magicFast(int[] array, int start, int end) {
2     if (end < start || start < 0 || end >= array.length) {
3         return -1;
4     }
5     int midIndex = (start + end) / 2;
6     int midValue = array[midIndex];
7     if (midValue == midIndex) {
8         return midIndex;
9     } success = false; // Установка флагом состояния для дальнейшего использования
10 }

```

```

11  /* Поиск влево */
12  int leftIndex = Math.min(midIndex - 1, midValue);
13  int left = magicFast(array, start, leftIndex);
14  if (left >= 0) {
15      return left;
16  }
17
18 /* Поиск вправо */
19  int rightIndex = Math.max(midIndex + 1, midValue);
20  int right = magicFast(array, rightIndex, end);
21
22  return right;
23 }
24
25 public static int magicFast(int[] array) {
26     return magicFast(array, 0, array.length - 1);
27 }

```

Этот код работает аналогично коду из предыдущего решения.

9.4. Напишите метод, возвращающий все подмножества одного множества.

Решение

Сначала определимся с разумными оценками времени и пространства. Сколько подмножеств будет? Мы можем сделать оценку, предполагая, что у каждого элемента есть «выбор» — находится в подмножестве или нет. Таким образом, у первого элемента есть два варианта — принадлежать подмножеству или не принадлежать. Пройдясь по всем элементам, мы получаем $\{2 * 2 * \dots\} 2^n$ или 2^n подмножеств. Поэтому нам не добиться лучшего результата, чем $O(2^n)$ времени/пространства.

Множества $\{a_1, a_2, \dots, a_n\}$ принято называть степенными и обозначать как $P(\{a_1, a_2, \dots, a_n\})$ или $P(n)$.

Решение 1: рекурсивное

Эта задача идеально подходит для метода «Базовый случай». Представьте, что мы пытаемся найти все подмножества множества $S = \{a_1, a_2, \dots, a_n\}$.

$n = 0$

Одно пустое подмножество: {}

$n = 1$

В $\{a_1\}$ есть два подмножества: {}, { a_1 }

$n = 2$

В $\{a_1, a_2\}$ есть 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.

Давайте сравним решения для $n = 3$ и $n = 2$:

$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$

$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$

Разница между этими решениями в том, что в $P(2)$ нет подмножеств, содержащих a_3 :

$$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

Как использовать $P(2)$, чтобы создать $P(3)$? Можно клонировать подмножества из $P(2)$ и добавить к ним a_3 :

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$

$$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

Объединив эти две строки, получаем $P(3)$.

$$n > 0$$

Построение решения $P(n)$ для общего случая — это обобщение приведенных выше шагов. Мы вычисляем $P(n-1)$, клонируем результаты и добавляем a_n в каждый из этих клонированных наборов.

```

1 ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set,
2 int index) {
3     ArrayList<ArrayList<Integer>> allsubsets;
4     if (set.size() == index) { // Базовый случай - добавляем пустое множество
5         allsubsets = new ArrayList<ArrayList<Integer>>();
6         allsubsets.add(new ArrayList<Integer>()); // Пустое множество
7     } else {
8         allsubsets = getSubsets(set, index + 1);
9         int item = set.get(index);
10        ArrayList<ArrayList<Integer>> moresubsets =
11            new ArrayList<ArrayList<Integer>>();
12        for (ArrayList<Integer> subset : allsubsets) {
13            ArrayList<Integer> newsubset = new ArrayList<Integer>();
14            newsubset.addAll(subset); //
15            newsubset.add(item);
16            moresubsets.add(newsubset);
17        }
18        allsubsets.addAll(moresubsets);
19    }
20    return allsubsets;
21 }
```

Это решение потребует $O(2^n)$ времени и пространства и будет наилучшим из всех возможных. Небольшую оптимизацию можно получить, если реализовать итерационную версию алгоритма.

Решение 2: комбинаторика

Предыдущее решение правильное, но мы рассмотрим еще один способ решить задачу.

Вспомните, что когда мы генерируем множество, 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 2^n 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.

Генерация всех подмножеств теперь превращается в получение всех двоичных (целых) чисел. Мы просто проходимся от 1 до 2^n и переводим двоичное представление числа в множество. Элементарно!

```

1 ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
2     ArrayList<ArrayList<Integer>> allsubsets =
3         new ArrayList<ArrayList<Integer>>();
4     int max = 1 << set.size(); /* Вычисляем  $2^n$  */
5     for (int k = 0; k < max; k++) {
6         ArrayList<Integer> subset = convertIntToSet(k, set);
7         allsubsets.add(subset);
8     }
9     return allsubsets;
10 }
11
12 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
13     ArrayList<Integer> subset = new ArrayList<Integer>();
14     int index = 0;
15     for (int k = x; k > 0; k >>= 1) {
16         if ((k & 1) == 1) {
17             subset.add(set.get(index));
18         }
19         index++;
20     }
21     return subset;
22 }
```

Это решение ничем не лучше и не хуже предыдущего.

9.5. Напишите метод, возвращающий все перестановки строки.

Решение

Подобно другим рекурсивным задачам, здесь мы будем использовать «Базовый случай». Предположим, что у нас есть строка S , состоящая из символов $a_1 a_2 \dots a_n$.

$n = 1$

Одна перестановка $S = a_1$ — это строка a_1

$n = 2$

Перестановками строки $S = a_1 a_2$ являются строки: $a_1 a_2$ и $a_2 a_1$

$n = 3$

Этот случай интереснее. Как сгенерировать все перестановки $a_1 a_2 a_3$, если известны перестановки $a_1 a_2$? То есть как получить:

$a_1 a_2 a_3$, $a_1 a_3 a_2$, $a_2 a_1 a_3$, $a_2 a_3 a_1$, $a_3 a_1 a_2$, $a_3 a_2 a_1$,

если мы имеем:

$a_1 a_2$, $a_2 a_1$

Разница между этими списками в том, что первый содержит a_3 , а второй — нет. Как же сгенерировать $f(3)$ из $f(2)$? Расставить a_3 во все возможные позиции строк $f(2)$.

$n > 0$

Для общего случая нам нужно просто повторить предыдущий процесс — находим $f(n-1)$ и затем расставляем a_n во все возможные позиции строк.

Следующий код выполняет эту задачу:

```

1 public static ArrayList<String> getPerms(String str) {
2     if (str == null) {
3         return null;
4     }
5     ArrayList<String> permutations = new ArrayList<String>();
6     if (str.length() == 0) { // базовый случай
7         permutations.add("");
8         return permutations;
9     }
10    int firstIndex = 0;
11    char first = str.charAt(0); // получаем первый символ
12    String remainder = str.substring(1); // удаляем первый символ
13    ArrayList<String> words = getPerms(remainder);
14    for (String word : words) {
15        for (int j = 0; j <= word.length(); j++) {
16            String s = insertCharAt(word, first, j);
17            permutations.add(s);
18        }
19    }
20    return permutations;
21 }
22
23 public static 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 }
```

Это решение потребует $O(n!)$ времени и $n!$ перестановок. Мы не сможем найти лучшее решение.

9.6. Реализуйте алгоритм, выводящий все корректные (правильно открытие и закрытие) комбинации пар круглых скобок

Пример

Ввод: 3

Вывод: ((())), ((())(), (())()), ()((())), ()()()

Решение

Первая мысль — использовать рекурсивный подход, который строит решение для $f(n)$, добавляя пары круглых скобок в $f(n-1)$. Это, конечно, правильная мысль.

Рассмотрим решение для $n = 3$:

((())) ((())) ()((()) ((())() ()()()

Как получить это решение из решения для $n = 2$?

(()) ()()

Можно расставить пары скобок в каждую существующую пару скобок, а также одну пару в начале строки. Другие места, куда мы могли вставить скобки, например в конце строки, получатся сами собой.

Итак, у нас есть 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 */
```

Но постойте! Некоторые пары дублируются! Стока `()()` упомянута дважды!

Если мы будем использовать данный подход, то нам понадобится проверка дубликатов перед добавлением строки в список.

```
1 public static 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) == '(') {
10                     String s = insertInside(str, i);
11                     if (!set.contains(s)) {
12                         set.add(s);
13                     }
14                 }
15             }
16             if (!set.contains("()" + str)) {
17                 set.add("()" + str);
18             }
19         }
20     }
21     return set;
22 }
23
24 public String insertInside(String str, int leftIndex) {
25     String left = str.substring(0, leftIndex + 1);
26     String right = str.substring(leftIndex + 1, str.length());
27     return left + "(" + right;
28 }
```

Алгоритм работает, но не очень эффективно. Мы тратим много времени на дублирующиеся строки.

Избежать проблемы дублирования можно путем построения строки с нуля. Этот подход подразумевает, что мы добавляем левые и правые скобки, пока наше выражение остается правильным.

При каждом рекурсивном вызове мы получаем индекс определенного символа в строке. Теперь нужно выбрать скобку (левую или правую). Когда использовать левую скобку, а когда — правую?

- Левая скобка: пока мы не израсходовали все левые скобки, мы можем вставить левую скобку.
- Правая скобка: мы можем добавить правую скобку, 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 левых скобок больше, чем правых, то вставляем правую скобку и продолжаем рекурсию.

```

1 public void addParen(ArrayList<String> list, int leftRem,
2 int rightRem, char[] str, int count) {
3     if (leftRem < 0 || rightRem < leftRem) return; // некорректное состояние
4
5     if (leftRem == 0 && rightRem == 0) { /* нет больше левых скобок */
6         String s = String.valueOf(str);
7         list.add(s);
8     } else {
9         /* Добавляем левую скобку, если остались любые левые скобки */
10        if (leftRem > 0) {
11            str[count] = '(';
12            addParen(list, leftRem - 1, rightRem, str, count + 1);
13        }
14
15        /* Добавляем правую скобку, если выражение верно */
16        if (rightRem > leftRem) {
17            str[count] = ')';
18            addParen(list, leftRem, rightRem - 1, str, count + 1);
19        }
20    }
21 }
22
23 public 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.

- 9.7.** Реализуйте функцию заливки краской, которая используется во многих графических редакторах. Данна плоскость (двумерный массив цветов), точка и цвет, которым нужно заполнить все окружающее пространство, окращенное в другой цвет.

Решение

Прежде всего, давайте визуализируем работу метода. Когда мы вызываем `paintFill` («нажимаем» кнопку заливки в графическом редакторе), находясь, например, на зеленом пикселе, то хотим расширить границы. Мы продвигаемся все дальше и дальше, вызывая `paintFill` для окружающих пикселов. Если цвет пикселя отличается от зеленого, мы останавливаемся.

Можно реализовать этот алгоритм рекурсивно:

```

1 enum Color {
2     Black, White, Red, Yellow, Green
3 }
4
5 boolean paintFill(Color[][] screen, int x, int y, Color ocolor,
6 Color ncolor) {
7     if (x < 0 || x >= screen[0].length ||
8         y < 0 || y >= screen.length) {
9         return false;
10    }
11    if (screen[y][x] == ocolor) {
12        screen[y][x] = ncolor;
13        paintFill(screen, x - 1, y, ocolor, ncolor); // left
14        paintFill(screen, x + 1, y, ocolor, ncolor); // right
15        paintFill(screen, x, y - 1, ocolor, ncolor); // top
16        paintFill(screen, x, y + 1, ocolor, ncolor); // bottom
17    }
18    return true;
19 }
20
21 boolean paintFill(Color[][] screen, int x, int y, Color ncolor){
22     return paintFill(screen, x, y, screen[y][x], ncolor);
23 }
```

Обратите внимание на порядок следования `x` и `y` в массиве `screen[y][x]` и запомните, что, когда вы решаете задачу, связанную с компьютерной графикой, нужно использовать именно такой порядок. 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 ежедневном программировании.

9.8. 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 рассчитать `makeChange(n)`, основываясь на предыдущих решениях (подзадачах). Пусть $n = 100$. Мы хотим вычислить количество способов представления 100 центов.

Нам известно, что для получения 100 центов мы можем использовать монеты 0, 1, 2, 3 или 4 четвертака (25 центов):

```
makeChange(100) =
    makeChange(100, используя 0 четвертаков) +
    makeChange(100, используя 1 четвертак) +
    makeChange(100, используя 2 четвертака) +
    makeChange(100, используя 3 четвертака) +
    makeChange(100, используя 4 четвертака)
```

Двигаемся дальше: попробуем упростить некоторые из этих задач. Например, `makeChange(100 используя 1 четвертак) = makeChange(75 используя 0 четвертаков)`. Это так, потому что если мы должны использовать один четвертак для представления 100 центов, оставшиеся варианты соответствуют различным представлениям 75 центов.

Мы можем применить эту же логику для `makeChange(100 используя 2 четвертака)`, `makeChange(100 используя 3 четвертака)` и `makeChange(100 используя 4 четвертака)`. Приведенное ранее выражение можно свести к следующему:

```
makeChange(100) =
    makeChange(100, используя 0 четвертаков) +
    makeChange(75, используя 0 четвертаков) +
    makeChange(50, используя 0 четвертаков) +
    makeChange(25, используя 0 четвертаков) +
    1
```

Заметьте, что последнее выражение — `makeChange(100 используя 4 четвертака)` — 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. Мы применим его для четырех частей приведенного выше выражения. Так, для первой части:

```
makeChange(100, используя 0 четвертаков) =
    makeChange(100, используя 0 четвертаков, 0 монет в 10 центов) +
    makeChange(100, используя 0 четвертаков, 1 монету в 10 центов) +
    makeChange(100, используя 0 четвертаков, 2 монеты в 10 центов) +
    ...
    makeChange(100, используя 0 четвертаков, 10 монет в 10 центов)
makeChange(75, используя 0 четвертаков) =
    makeChange(75, используя 0 четвертаков, 0 монет в 10 центов) +
    makeChange(75, используя 0 четвертаков, 1 монету в 10 центов) +
    makeChange(75, используя 0 четвертаков, 2 монеты в 10 центов) +
    ...
    makeChange(75, используя 0 четвертаков, 7 монет в 10 центов)
makeChange(50, используя 0 четвертаков) =
    makeChange(50, используя 0 четвертаков, 0 монет в 10 центов) +
    makeChange(50, используя 0 четвертаков, 1 монету в 10 центов) +
    makeChange(50, используя 0 четвертаков, 2 монеты в 10 центов) +
    ...
    makeChange(50, используя 0 четвертаков, 5 монет в 10 центов)
makeChange(25, используя 0 четвертаков) =
    makeChange(25, используя 0 четвертаков, 0 монет в 10 центов) +
    makeChange(25, используя 0 четвертаков, 1 монету в 10 центов) +
    makeChange(25, используя 0 четвертаков, 2 монеты 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, используя 0 четвертаков, 5 монет в 10 центов)` полностью сводится к 1, так как 5 монет по 10 центов дает ровно 50 центов.

Рекурсивный алгоритм будет иметь примерно такой вид:

```

1 public int makeChange(int n, int denom) {
2     int next_denom = 0;
3     switch (denom) {
4         case 25:
5             next_denom = 10;
6             break;
7         case 10:
8             next_denom = 5;
9             break;
10        case 5:
11            next_denom = 1;
12            break;
13        case 1:
14            return 1;
15    }
16
17    int ways = 0;
18    for (int i = 0; i * denom <= n; i++) {
19        ways += makeChange(n - i * denom, next_denom);
20    }
21    return ways;
22 }
23
24 System.out.writeln(makeChange(100, 25));

```

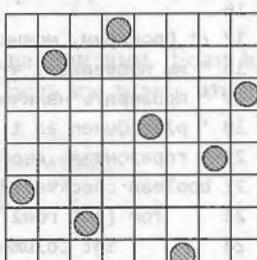
Хотя мы реализовали код, опираясь на монеты, используемые в США, его можно легко адаптировать для любой другой валюты.

- 9.9.** *Дана шахматная доска размером 8×8. Напишите алгоритм, находящий все варианты расстановки восьми ферзей так, чтобы никакие две фигуры не попадали на одну горизонталь, вертикаль или диагональ. Учитываются не только главные, но и все остальные диагонали.*

Решение

У нас 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.

Давайте поместим последнего ферзя на 8-й горизонтали (ведь сейчас расположение ферзей не важно). На какой клетке находится ферзь? Существует восемь вариантов — по одному для каждой вертикали.



Решение задачи о восьми ферзях

Так, мы хотим знать все варианты размещения 8 ферзей на доске размером 8×8:

Число способов расположить 8 ферзей на доске 8×8 =
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 0) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 1) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 2) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 3) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 4) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 5) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 6) +
 способов расположить 8 ферзей на доске 8×8 с ферзем на клетке (7, 7) +

Мы можем вычислить каждое слагаемое, используя простой способ:

Число способов расположить 8 ферзей на доске 8×8 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, что нам не нужно учитывать комбинации с ферзями на клетках (7, 3) и (6, 3), поскольку они нарушают требование, что каждый ферзь должен находиться на собственной вертикали/горизонтали/диагонали.

Теперь решение задачи будет предельно простым:

```

1 int GRID_SIZE = 8;
2
3 void placeQueens(int row, Integer[] columns,
4 ArrayList<Integer[]> results) {
5     if (row == GRID_SIZE) { // Найдено верное расположение
6         results.add(columns.clone());
7     } else {
8         for (int col = 0; col < GRID_SIZE; col++) {
9             if (checkValid(columns, row, col)) {
10                 columns[row] = col; // Помещаем ферзя
11                 placeQueens(row + 1, columns, results);
12             }
13         }
14     }
15 }
16
17 /* Проверим, можно ли в (row1, column1) разместить ферзя. Для этого
18 * мы проверяем, чтобы не было ферзей на той же вертикали или диагонали. Не нужно
19 * проверять наличие ферзя на этой же горизонтали. При вызове
20 * placeQueen за 1 раз размещается только 1 ферзь. Мы знаем, что
21 * горизонталь свободна. */
22 boolean checkValid(Integer[] columns, int row1, int column1) {
23     for (int row2 = 0; row2 < row1; row2++) {
24         int column2 = columns[row2];
25         /* Проверяем если (row2, column2) лишает (row1, column1) места
26         * для ферзя */

```

```

27
28     /* Проверяем, наличие ферзя на той же вертикали */
29     if (column1 == column2) {
30         return false;
31     }
32     /* Проверяем диагонали: если дистанция между вертикалями равна
33     * дистанции между горизонтальами, значит, они на
34     * одной диагонали. */
35     int columnDistance = Math.abs(column2 - column1);
36
37     /* row1 > row2, мы не нуждаемся в abs */
38     int rowDistance = row1 - row2;
39     if (columnDistance == rowDistance) {
40         return false;
41     }
42 }
43 return true;
44 }
45 }

```

Так как на каждой горизонтали может быть только один ферзь, нам не нужно хранить нашу доску как полную матрицу 8×8 . Достаточно одномерного массива, где $\text{column}[row] = c$ указывает, что на горизонтали r расположен ферзь, занимающий вертикаль c .

- 9.10.** Дан штабель из n ящиков шириной w_i , высотой h_i и глубиной d_i . Ящики нельзя поворачивать, добавлять ящики можно только наверх штабеля. Каждый нижний ящик в стопке по высоте, ширине и глубине больше ящика, который находится на нем. Реализуйте метод, позволяющий построить самый высокий штабель (высота штабеля равна сумме высот всех ящиков).

Решение

Чтобы решить эту задачу, нужно найти связи между различными подзадачами.

Предположим, что у нас были ящики: b_1, b_2, \dots, b_n . Наибольший стек (стопка), который мы можем создать (со всеми ящиками), эквивалентен максимуму из (наибольший стек с основанием b_1 , наибольший стек с основанием b_2, \dots , наибольший стек с основанием b_n).

Но как найти наибольший стек с конкретным основанием? Можно использовать тот же подход — поэкспериментировать 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 стек $\{b_1, b_5, \dots\}$. Ведь мы знаем, что b_1 не может находиться под b_5 .

Приведенный далее код реализует данный алгоритм в рекурсивном виде.

```

1 public ArrayList<Box> createStackR(Box[] boxes, Box bottom) {
2     int max_height = 0;
3     ArrayList<Box> max_stack = null;
4     for (int i = 0; i < boxes.length; i++) {

```

продолжение ↗

```

5         if (boxes[i].canBeAbove(bottom)) {
6             ArrayList<Box> new_stack = createStackR(boxes, boxes[i]);
7             int new_height = stackHeight(new_stack);
8             if (new_height > max_height) {
9                 max_stack = new_stack;
10                max_height = new_height;
11            }
12        }
13    }
14
15    if (max_stack == null) {
16        max_stack = new ArrayList<Box>();
17    }
18    if (bottom != null) {
19        max_stack.add(0, bottom); // Вставляем низ стека
20    }
21
22    return max_stack;
23 }

```

Проблема этого кода в том, что он неэффективен. Мы пытаемся найти лучшее решение, которое похоже на $\{b_3, b_4, \dots\}$, даже в том случае, когда нашли лучшее решение с b_4 в основании. Вместо того чтобы генерировать решения с нуля, можно кэшировать результаты, используя динамическое программирование.

```

1 public ArrayList<Box> createStackDP(Box[] boxes, Box bottom,
2     HashMap<Box, ArrayList<Box>> stack_map) {
3     if (bottom != null && stack_map.containsKey(bottom)) {
4         return stack_map.get(bottom);
5     }
6
7     int max_height = 0;
8     ArrayList<Box> max_stack = null;
9     for (int i = 0; i < boxes.length; i++) {
10        if (boxes[i].canBeAbove(bottom)) {
11            ArrayList<Box> new_stack =
12                createStackDP(boxes, boxes[i], stack_map);
13            int new_height = stackHeight(new_stack);
14            if (new_height > max_height) {
15                max_stack = new_stack;
16                max_height = new_height;
17            }
18        }
19    }
20
21    if (max_stack == null) max_stack = new ArrayList<Box>();
22    if (bottom != null) max_stack.add(0, bottom);
23    stack_map.put(bottom, max_stack);
24
25    return (ArrayList<Box>)max_stack.clone();
26 }

```

Вы можете поинтересоваться, почему в строке 25 выполняется приведение типа `max_stack.clone()`. Неужели `max_stack` содержит некорректные данные? Это так, но мы нуждаемся в приведении типа.

Метод `clone()` был объявлен в классе `Object`:

```
1 protected Object clone() { ... }
```

Когда мы переопределяем метод, то изменяем параметры, но не изменяем тип возвращаемого значения. Поэтому если `Foo` наследуется от `Object` и переопределяет `clone()`, то `clone()` возвратит экземпляр типа `Object`.

Именно это и происходит с оператором `(ArrayList<Box>)max_stack.clone()`. Класс стека переопределяет метод `clone()`, но он все еще возвращает `Object`. Так что мы должны выполнить операцию приведения для возвращаемого значения.

- 9.11.** Дано логическое выражение, построенное из символов 0, 1, &, | и ^ и имеющее значение `result`. Напишите функцию, подсчитывающую количество вариантов логических выражений, дающих значение `result`.

Пример

Выражение: `1^0|0|1`

Результат: `result = false (0)`

Вывод: 2 способа: `1^((0|0)|1)` и `1^(0|(0|1))`

Решение

Как в других рекурсивных задачах, ключ к решению — выяснить связь между задачей и подзадачами.

Предположим, что `int f(expression, result)` — это функция, возвращающая количество всех допустимых выражений, которые равны результату. Мы хотим найти `f(1^0|0|1, true)` (то есть все способы расставить скобки в выражении `1^0|0|1` так, чтобы значение выражения было равно `true`). Таким образом:

```
f(1^0|0|1, true) = f(1 ^ (0|0|1), true) +
    f((1^0) | (0|1), true) +
    f((1^0|0) | 1, true)
```

Значит, мы можем пройтись по выражению, обрабатывая каждый оператор как первый оператор, который нужно заключить в скобки.

Теперь рассмотрим одно из внутренних выражений, например `f((1^0)|(0|1), true)`. Чтобы выражение принимало значение `true`, левая или правая его часть должны быть `true`. Поэтому выражение можно разбить на части следующим образом:

```
f((1^0) | (0|1), true) = f(1^0, true) * f(0|1, true) +
    f(1^0, false) * f(0|1, true) +
    f(1^0, true) * f(0|1, false)
```

Давайте сделаем аналогичную разбивку для каждого из булевых операторов:

```
f(exp1 | exp2, true) = f(exp1, true) * f(exp2, true) +
    f(exp1, true) * f(exp2, false) +
    f(exp1, false) * f(exp2, true)
f(exp1 & exp2, true) = f(exp1, true) * f(exp2, true)
f(exp1 ^ exp2, true) = f(exp1, true) * f(exp2, false) +
    f(exp1, false) * f(exp2, true)
```

Для `false` также можно провести подобную операцию:

$$\begin{aligned} f(exp1 \mid exp2, \text{false}) &= f(exp1, \text{false}) * f(exp2, \text{false}) \\ f(exp1 \& exp2, \text{false}) &= f(exp1, \text{false}) * f(exp2, \text{false}) + \\ &\quad f(exp1, \text{true}) * f(exp2, \text{false}) + \\ &\quad f(exp1, \text{false}) * f(exp2, \text{true}) \\ f(exp1 ^ exp2, \text{false}) &= f(exp1, \text{true}) * f(exp2, \text{true}) + \\ &\quad f(exp1, \text{false}) * f(exp2, \text{false}) \end{aligned}$$

Реализация кода является теперь делом техники. (Чтобы избежать лишнего переноса строк и упростить код, мы дали переменным короткие имена.)

```

1  public int f(String exp, boolean result, int s, int e) {
2      if (s == e) {
3          if (exp.charAt(s) == '1' && result) {
4              return 1;
5          } else if (exp.charAt(s) == '0' && !result) {
6              return 1;
7          }
8          return 0;
9      }
10     int c = 0;
11     if (result) {
12         for (int i = s + 1; i <= e; i += 2) {
13             char op = exp.charAt(i);
14             if (op == '&') {
15                 c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
16             } else if (op == '|') {
17                 c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
18                 c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
19                 c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
20             } else if (op == '^') {
21                 c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
22                 c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
23             }
24         }
25     } else {
26         for (int i = s + 1; i <= e; i += 2) {
27             char op = exp.charAt(i);
28             if (op == '&') {
29                 c += f(exp, false, s, i - 1) * f(exp, true, i + 1, e);
30                 c += f(exp, true, s, i - 1) * f(exp, false, i + 1, e);
31                 c += f(exp, false, s, i - 1) * f(exp, false, i + 1, e);
32             } else if (op == '|') {
33                 c += f(exp, false, s, i - 1) * f(exp, false, i + 1, e);
34             } else if (op == '^') {
35                 c += f(exp, true, s, i - 1) * f(exp, true, i + 1, e);
36                 c += f(exp, false, s, i - 1) * f(exp, false, i + 1, e);
37             }
38         }
39     }
40     return c;
41 }
```

Алгоритм работает, код не эффективен. Этот метод много раз вычисляет $f(exp)$ для одного и того же значения exp .

Чтобы избавиться от этой проблемы, можно использовать динамическое программирование и кэшировать результаты. Давайте будем кэшировать `expression` (выражение) и `result` (результат).

```

1 public int f(String exp, boolean result, int s, int e,
2   HashMap<String, Integer> q) {
3     String key = "" + result + s + e;
4     if (q.containsKey(key)) {
5       return q.get(key);
6     }
7     if (s == e) {
8       if (exp.charAt(s) == '1' && result == true) {
9         return 1;
10      } else if (exp.charAt(s) == '0' && result == false) {
11        return 1;
12      }
13    } else if (exp.charAt(s) == '&') {
14      return 0;
15    }
16    int c = 0;
17    if (result) {
18      for (int i = s + 1; i <= e; i += 2) {
19        char op = exp.charAt(i);
20        if (op == '&') {
21          c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
22        } else if (op == '|') {
23          c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
24          c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
25          c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
26        } else if (op == '^') {
27          c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
28          c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
29        }
30      }
31    } else {
32      for (int i = s + 1; i <= e; i += 2) {
33        char op = exp.charAt(i);
34        if (op == '&') {
35          c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
36          c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q); } else {
37            c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q);
38          } else if (op == '|') {
39            c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q); } else {
40          } else if (op == '^') {
41            c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q); } else {
42            c += f(exp,false,s,i-1,q) * f(exp,false,i+1,e,q); } else {
43          }
44        }
45      }
}

```

продолжение ↗

```

46     q.put(key, c);
47     return c;
48 }

```

Теперь код оптимизирован, но не полностью. Если бы мы знали, сколько существует способов заключить выражение в скобки, то могли бы рассчитать $f(\text{exp} = \text{false})$ как $\text{total}(\text{exp}) - f(\text{exp} = \text{true})$.

Можно воспользоваться аналитической формулой, позволяющей рассчитать количество вариантов расстановки скобок в выражении, но, скорее всего, вы с ней не знакомы, — это числовая последовательность Каталана:

$$C = \frac{2n!}{(n+1)!n!}$$

С этим уточнением наш код примет вид:

```

1 public int f(String exp, boolean result, int s, int e,
2   HashMap<String, Integer> q) {
3   String key = "" + s + e;
4   int c = 0;
5   if (!q.containsKey(key)) {
6     if (s == e) {
7       if (exp.charAt(s) == '1') c = 1;
8       else c = 0;
9     }
10    for (int i = s + 1; i <= e; i += 2) {
11      char op = exp.charAt(i);
12      if (op == '&') {
13        c += f(exp,true,s,i-1,q) * f(exp,true,i+1,e,q);
14      } else if (op == '|') {
15        int left_ops = (i-1-s)/2; // скобки слева
16        int right_ops = (e - i - 1) / 2; // скобки справа
17        int total_ways = total(left_ops) * total(right_ops);
18        int total_false = f(exp,false,s,i-1,q) *
19          f(exp,false,i+1,e,q);
20        c += total_ways - total_false;
21      } else if (op == '^') {
22        c += f(exp,true,s,i-1,q) * f(exp,false,i+1,e,q);
23        c += f(exp,false,s,i-1,q) * f(exp,true,i+1,e,q);
24      }
25    }
26    q.put(key, c);
27  } else {
28    c = q.get(key);
29  }
30  if (result) {
31    return c;
32  } else {
33    int num_ops = (e - s) / 2;
34    return total(num_ops) - c;
35  }
36 }
37 }

```

10. Сортировка и поиск

- 10.1.** Дано: два отсортированных массива — A и B. Размер массива A (свободное место в конце) позволяет поместить в него массив B. Напишите метод слияния массивов B и A, сохраняющий сортировку.

Решение

Так как мы знаем, что массив A имеет достаточно места в конце, нет необходимости выделять дополнительное пространство. Наша 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 сдвигать остальные элементы, чтобы появилось 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 public static void merge(int[] a, int[] b, int lastA, int lastB) {
2     int indexA = lastA - 1; /* Индекс последнего элемента в массиве b */
3     int indexB = lastB - 1; /* Индекс последнего элемента в массиве a */
4     int indexMerged = lastB + lastA - 1; /* конец объединенного массива */
5
6     /* Объединяем a и b, начиная с последних элементов */
7     while (indexA >= 0 && indexB >= 0) {
8         /* конец a > конец b */
9         if (a[indexA] > b[indexB]) {
10             a[indexMerged] = a[indexA]; // копируем элементы
11             indexMerged--; // сдвигаем индексы
12             indexA--;
13         } else {
14             a[indexMerged] = b[indexB]; // копируем элементы
15             indexMerged--; // сдвигаем индексы
16             indexB--;
17         }
18     }
19
20     /* Копируем оставшиеся элементы из b */
21     while (indexB >= 0) {
22         a[indexMerged] = b[indexB];
23         indexMerged--;
24         indexB--;
25     }
26 }
```

Обратите внимание, что вам не придется копировать содержимое массива A после того, как все элементы из B будут расставлены. 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, что вам не требуется придерживаться какого-либо определенного порядка слов.

Один из способов решить задачу — использовать любой стандартный алгоритм сортировки, например сортировку слиянием или быструю сортировку, но изменить алгоритм сравнения (компаратор). Компаратор позволяет выяснить, что две анаграммы являются эквивалентными.

Вы знаете самый простой способ проверить, являются ли слова анаграммами? Можно подсчитать частоту использования разных символов в каждой строке и возвратить `true`, если эти значения совпадают. С другой стороны, можно отсортировать строки, после чего строки-анаграммы станут одинаковыми.

Приведенный далее код реализует компаратор:

```

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 сделать сортировку массива, используя метод `compareTo` вместо обычного:

```
12 Arrays.sort(array, new AnagramComparator());
```

Этот алгоритм занимает $O(n \log(n))$ времени.

Это лучший возможный результат, который дает общий алгоритм сортировки, но на самом деле нам не нужно сортировать массив полностью. Нам нужно *сгруппировать* строки в массиве в соответствии с анаграммами.

Это можно осуществить с помощью хэш-таблицы, которая устанавливает соответствие между отсортированными словами и анаграммами. Так, например, `acre` будет соответствовать списку слов `{acre, race, care}`. Как только мы сгруппируем слова по анаграммам, то сможем поместить их в обратно в массив.

Приведенный далее код реализует этот алгоритм:

```

1 public void sort(String[] array) {
2     Hashtable<String, LinkedList<String>> hash =
3         new Hashtable<String, LinkedList<String>>();
4
5     /* Группируем слова по анаграммам */
6     for (String s : array) {
7         String key = sortChars(s);
```

```

8     if (!hash.containsKey(key)) {
9         hash.put(key, new LinkedList<String>());
10    }
11    LinkedList<String> anagrams = hash.get(key);
12    anagrams.push(s);
13 }
14
15 /* Конвертируем хэш-таблицу в массив */
16 int index = 0;
17 for (String key : hash.keySet()) {
18     LinkedList<String> list = hash.get(key);
19     for (String t : list) {
20         array[index] = t;
21         index++;
22     }
23 }
24 }

```

Обратите внимание, что этот алгоритм представляет собой модификацию алгоритма блочной сортировки.

- 10.3.** Дано: отсортированный массив из n целых чисел, который был циклически сдвинут произвольное количество раз. Напишите код для поиска элемента в массиве. Исходите из предположения, что массив изначально был отсортирован по возрастанию.

Решение

Если вы думаете, что для решения этой задачи можно воспользоваться бинарным поиском, вы правы!

В классической задаче бинарного поиска мы сравниваем x с центральной точкой, чтобы узнать, куда поместить 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 мы проанализируем задачу, то обратим внимание, что половина массива отсортирована по возрастанию. Можно взглянуть на отсортированную половину массива и принять решение, 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, — например {2, 2, 2, 3, 4, 2}. В этом случае можно выполнить сравнение с правым элементом и, если он отличается, искать только в правой половине. Если нам не повезло, то придется анализировать обе половины.

```

1  public int search(int a[], int left, int right, int x) {
2      int mid = (left + right) / 2;
3      if (x == a[mid]) { // Найден элемент
4          return mid;
5      }
6      if (right < left) {
7          return -1;
8      }
9
10     /* Левая или правая половины могут быть отсортированы. Нужно узнать, какая
11     * часть отсортирована нормально, и затем использовать
12     * отсортированную половину, чтобы найти x. */
13     if (a[left] < a[mid]) { // Левая половина отсортирована
14         if (x >= a[left] && x <= a[mid]) {
15             return search(a, left, mid - 1, x); // Поиск в левой половине
16         } else {
17             return search(a, mid + 1, right, x); // Поиск в правой половине
18         }
19     } else if (a[mid] < a[left]) { // Правая половина отсортирована
20         if (x >= a[mid] && x <= a[right]) {
21             return search(a, mid + 1, right, x); // Поиск в правой половине
22         } else {
23             return search(a, left, mid - 1, x); // Поиск в левой половине
24         }
25     } else if (a[left] == a[mid]) { // Левая половина - все повторения
26         if (a[mid] != a[right]) { // Если правая отличается, поиск в ней
27             return search(a, mid + 1, right, x); // Поиск в правой половине
28         } else { // Иначе нужно произвести поиск в обеих половинах
29             int result = search(a, left, mid - 1, x); // Поиск в левой половине
30             if (result == -1) {
31                 return search(a, mid + 1, right, x); // Поиск в правой половине
32             } else {
33                 return result;
34             }
35         }
36     }
37     return -1;
38 }
```

Код потребует $O(\log n)$ времени, если все элементы будут разными. Если в массиве окажется много повторяющихся элементов, алгоритм потребует $O(n)$ времени. Когда в массиве содержится много повторяющихся элементов, часто приходится обрабатывать обе половины (левую и правую) массива.

Данная задача не сложна, но написать идеальную реализацию достаточно трудно. Не беспокойтесь, если у вас возникнут проблемы при решении этой задачи. Вы должны тщательно протестировать код.

10.4. Дано: файл размером 20 Гбайт, содержащий строки. Как выполнить сортировку такого файла?

Решение

Если интервьюер установил ограничение 20 Гбайт, вы должны задуматься. Скорее всего, интервьюер не хочет, чтобы вы загружали в память все данные.

Что же делать? Нам придется загружать в память только часть данных.

Давайте разделим файл на блоки по x Мбайт каждый, где x — размер памяти, которую вы можете использовать. Каждый блок будет отдельно отсортирован и сохранен в файл.

Как только вы отсортируете все блоки, можно приступать к объединению блоков. В результате в вашем распоряжении окажется полностью отсортированный файл.

Данный алгоритм получил название «внешняя сортировка».

10.5. Дано: отсортированный массив, состоящий из строк, разделенных пустыми строками. Напишите метод для обнаружения позиции заданной строки.

Решение

Пустые строки мешают нам использовать бинарный поиск и сравнивать строку, которую мы ищем (`str`), со средним элементом массива.

Строки-разделители заставляют нас реализовывать модифицированную версию бинарного поиска. Необходимо исправить сравнение с `mid`, если `mid` оказывается пустой строкой. Для этого достаточно переместить `mid` на ближайшую непустую строку.

Приведенный далее рекурсивный код решает эту задачу. Впрочем, данное решение можно легко преобразовать в итеративное. Полную версию этого кода можно скачать с сайта автора книги.

```

1 public int searchR(String[] strings, String str, int first,
2 int last) {
3     /* Помещаем mid в середину */
4     int mid = (last + first) / 2;
5
6     /* Если mid - пустая строка, находим ближайшую непустую строку */
7     if (strings[mid].isEmpty()) {
8         int left = mid - 1;
9         int right = mid + 1;
10        while (true) {
11            if (left < first && right > last) {
12                return -1;
13            } else if (right <= last && !strings[right].isEmpty()) {
14                mid = right;
15                break;
16            } else if (left >= first && !strings[left].isEmpty()) {
17                mid = left;
18                break;
19            }
20            right++;
21            left--;
}

```

продолжение ↴

```

22     }
23 }
24
25 /* Проверяем строку и делаем рекурсивный вызов в случае необходимости */
26 if (str.equals(strings[mid])) { // Найдена!
27     return mid;
28 } else if (strings[mid].compareTo(str) < 0) { // Поиск вправо
29     return searchR(strings, str, mid + 1, last);
30 } else { // Поиск влево
31     return searchR(strings, str, first, mid - 1);
32 }
33 }
34
35 public int search(String[] strings, String str) {
36     if (strings == null || str == null || str == "") {
37         return -1;
38     }
39     return searchR(strings, str, 0, strings.length - 1);
40 }

```

Будьте внимательны, работая с пустыми строками. Задумайтесь, нужно ли вам находить расположение пустой строки? (Это потребует $O(n)$ времени.) Или достаточно обработать эту ситуацию как ошибку?

Не существует правильного ответа на эти вопросы. Эту проблему необходимо обсудить с интервьюером. Если вы поинтересуетесь, что делать с пустыми строками, то покажете себя аккуратным и предусмотрительным программистом.

- 10.6.** Дано: матрица размером $M \times N$, строки и столбцы которой отсортированы.
Напишите метод, находящий указанный элемент.

Решение

Чтобы найти нужный элемент, можно воспользоваться бинарным поиском по каждой строке. Алгоритм потребует $O(M \log(N))$ времени, так как необходимо обработать M столбцов, на каждый из которых тратится $O(\log(N))$ времени. Задайте уточняющий вопрос интервьюеру прежде, чем приступите к решению.

Прежде чем приступить к разработке алгоритма, давайте 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 мы посмотрим на первые элементы строки и столбца, то можем 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 проверку 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 так, поскольку последний элемент всегда будет максимальным.

Давайте используем все эти наблюдения для построения решения:

- Если первый элемент столбца больше x , то x находится в колонке слева.
- Если последний элемент столбца меньше x , то x находится в колонке 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.

Давайте начнем со столбцов.

Мы должны начать с правого столбца и двигаться влево. Это означает, что первым элементом для сравнения будет $[0][c-1]$, 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. Давайте начнем с $[0][2]$.

Данный элемент может не являться последним элементом строки в полной матрице, но это — 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 отброшены):

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Мы можем раз за разом использовать наши правила поиска. Обратите внимание, что мы используем правила 1 и 4.

Следующий код реализует этот алгоритм:

```

1 public static 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 }
```

Другой подход к решению задачи — бинарный поиск. Мы получим более сложный код, но построен он будет на тех же правилах.

Решение 2: бинарный поиск

Давайте еще раз обратимся к нашему примеру.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

Мы хотим повысить эффективность алгоритма. Давайте зададимся вопросом: где может находиться элемент?

Нам сказано, что все строки и столбцы отсортированы. Это означает, что элемент $[i][j]$ больше, чем элементы в строке i , находящиеся между столбцами 0 и j и элементы в строке j между строками 0 и $i-1$.

Другими словами:

$a[i][0] \leq a[i][1] \leq \dots \leq a[i][j-1] \leq a[i][j]$
 $a[0][j] \leq a[1][j] \leq \dots \leq a[i-1][j] \leq a[i][j]$

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 левого элемента, так и элемента, находящегося выше. Таким образом, выделенный элемент больше всех элементов, находящихся в квадрате.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

Можно сформулировать правило: нижний правый угол любого прямоугольника, выделенного в матрице, будет содержать самый большой элемент.

Аналогично, 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

Давайте вернемся к исходной задаче. Допустим, что нам нужно найти элемент 85. Если мы посмотрим на диагональ, то увидим элементы 35 и 95. Какую информацию о местонахождении элемента 85 можно из этого извлечь?

15	28	78	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.

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 можем эффективно использовать бинарный поиск.

Приведенный ниже код реализует этот алгоритм:

```

1 public Coordinate findElement(int[][] matrix, Coordinate origin,
2 Coordinate dest, int x) {
3     if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
4         return null;
5     }
6     if (matrix[origin.row][origin.column] == x) {
7         return origin;
8     } else if (!origin.isBefore(dest)) {
9         return null;
10    }
11
12    /* Установим start на начало диагонали, а end - на конец
13     * диагонали. Так как сетка, возможно, не является квадратной, конец
14     * диагонали может не равняться dest. */
15    Coordinate start = (Coordinate) origin.clone();
16    int diagDist = Math.min(dest.row - origin.row,
17        dest.column - origin.column);
18    Coordinate end = new Coordinate(start.row + diagDist,
19        start.column + diagDist);
20    Coordinate p = new Coordinate(0, 0);
21
22    /* Производим бинарный поиск по диагонали, ищем первый
23     * элемент больше x */
24    while (start.isBefore(end)) {
25        p.setToAverage(start, end);
26        if (x > matrix[p.row][p.column]) {
27            start.row = p.row + 1;
28            start.column = p.column + 1;
29        } else {

```

продолжение ↗

```
30         end.row = p.row - 1;
31         end.column = p.column - 1;
32     }
33 }
34
35 /* Разделяем сетку на квадранты. Ищем в нижнем левом и верхнем
36 * правом квадранте */
37 return partitionAndSearch(matrix, origin, dest, start, x);
38 }
39
40 public Coordinate partitionAndSearch(int[][] matrix,
41 Coordinate origin, Coordinate dest, Coordinate pivot,
42 int elem) {
43     Coordinate lowerLeftOrigin =
44         new Coordinate(pivot.row, origin.column);
45     Coordinate lowerLeftDest =
46         new Coordinate(dest.row, pivot.column - 1);
47     Coordinate upperRightOrigin =
48         new Coordinate(origin.row, pivot.column);
49     Coordinate upperRightDest =
50         new Coordinate(pivot.row - 1, dest.column);
51
52     Coordinate lowerLeft =
53         findElement(matrix, lowerLeftOrigin, lowerLeftDest, elem);
54     if (lowerLeft == null) {
55         return findElement(matrix, upperRightOrigin,
56             upperRightDest, elem);
57     }
58     return lowerLeft;
59 }
60
61 public static Coordinate findElement(int[][] matrix, int x) {
62     Coordinate origin = new Coordinate(0, 0);
63     Coordinate dest = new Coordinate(matrix.length - 1,
64         matrix[0].length - 1);
65     return findElement(matrix, origin, dest, x);
66 }
67
68 public class Coordinate implements Cloneable {
69     public int row;
70     public int column;
71     public Coordinate(int r, int c) {
72         row = r;
73         column = c;
74     }
75
76     public boolean inbounds(int[][] matrix) {
77         return row >= 0 && column >= 0 &&
78             row < matrix.length && column < matrix[0].length;
79     }
80
81     public boolean isBefore(Coordinate p) {
```

```

82     return row <= p.row && column <= p.column;
83 }
84
85 public Object clone() {
86     return new Coordinate(row, column);
87 }
88
89 public void setToAverage(Coordinate min, Coordinate max) {
90     row = (min.row + max.row) / 2;
91     column = (min.column + max.column) / 2;
92 }
93 }

```

Если вы посмотрели на этот код и подумали: «Я не смогу написать 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 следует думать, что вы находитесь в заведомо проигрышной ситуации, если вам досталась такая сложная задача.

Вы облегчите себе жизнь, выделяя код в методы. Например, можно сделать `partitionAndSearch` методом. Попробуйте сконцентрироваться на ключевых местах программы. Позже вы вернетесь к `partitionAndSearch` и напишете его тело, если останется время.

- 10.7. Цирк готовит новый аттракцион — пирамида из людей, стоящих на плечах друг у друга. Простая логика подсказывает, что люди, стоящие выше, должны быть ниже ростом и легче, чем люди, находящиеся в нижних ярусах пирамиды. Учитывая информацию о росте и весе каждого человека, напишите метод, вычисляющий наибольшее число человек в пирамиде.**

Решение

Если убрать всю «воду» из формулировки, то задача сводится к следующему.

У нас есть список с парами элементов. Нужно найти самую длинную последовательность элементов списка, такую, чтобы и первые и вторые элементы находились в «неубменьшающемся» порядке.

Если использовать подход «Упростить и обобщить» (или «Сопоставление с образцом»), можно связать эту задачу с обнаружением самой длинной увеличивающейся последовательности в массиве.

Подзадача: наибольшая увеличивающаяся подпоследовательность

Если элементы не должны оставаться в том же самом (относительном) порядке, можно отсортировать массив. Но это делает задачу слишком тривиальной, так что давайте предположим, что все элементы должны остаться на тех же местах.

Попытаемся получить рекурсивный алгоритм, анализируя массив поэлементно. Обратите внимание, что знание самой длинной увеличивающейся подпоследовательности от [0] до [i] не даст нам информации относительно элементов [i+1] и [i+2]. Рассмотрим простой пример.

Массив: 13, 14, 10, 11, 12
`Longest(0 - 0): 13`
`Longest(0 - 1): 13, 14`

продолжение ↴

```

Longest(0 - 2): 13, 14
Longest(0 - 3): 13, 14 OR 10, 11
Longest(0 - 4): 10, 11, 12

```

Если мы пытаемся найти только `Longest(0 - 4)` и `Longest(0 - 3)`, то не добьемся оптимального решения.

Попробуем использовать другой рекурсивный подход. Вместо того чтобы искать самую длинную увеличивающуюся подпоследовательность от 0 до i , можно найти самую длинную последовательность, которая заканчивается элементом i . Используя приведенный ранее пример, можно сделать следующее:

```

Массив: 13, 14, 10, 11, 12
Longest(ending with A[0]): 13
Longest(ending with A[1]): 13, 14
Longest(ending with A[2]): 10
Longest(ending with A[3]): 10, 11
Longest(ending with A[4]): 10, 11, 12

```

Обратите внимание, что самая длинная последовательность, заканчивающаяся $A[i]$, может быть найдена на основании всех предыдущих решений. Мы просто добавляем $A[i]$ к самой длинной допустимой последовательности (под допустимой последовательностью понимается любой список, где $A[i] > list.tail$).

Реальная задача: наибольшая увеличивающаяся последовательность пар

Теперь, когда мы знаем, как найти самую длинную увеличивающуюся подпоследовательность массива целых чисел, можно приступить к решению настоящей задачи. Давайте отсортируем список людей по росту и затем применим алгоритм `longestIncreasingSubsequence` только для веса.

Приведенный далее код реализует этот алгоритм:

```

1  ArrayList<HtWt> getIncreasingSequence(ArrayList<HtWt> items) {
2      Collections.sort(items);
3      return longestIncreasingSubsequence(items);
4  }
5
6  void longestIncreasingSubsequence(ArrayList<HtWt> array,
7  ArrayList<HtWt>[] solutions, int current_index) {
8      if (current_index >= array.size() || current_index < 0) return;
9      HtWt current_element = array.get(current_index);
10
11     /* Находим наибольшую последовательность, к которой
12        можно добавить current_element */
13     ArrayList<HtWt> best_sequence = null;
14     for (int i = 0; i < current_index; i++) {
15         if (array.get(i).isBefore(current_element)) {
16             best_sequence = seqWithMaxLength(best_sequence,
17             solutions[i]);
18         }
19     }
20     /* Добавляем current_element */
21     ArrayList<HtWt> new_solution = new ArrayList<HtWt>();
22     if (best_sequence != null) {

```

```

23     new_solution.addAll(best_sequence);
24 }
25 new_solution.add(current_element);
26
27 /* Добавляем в список, рекурсия */
28 solutions[current_index] = new_solution;
29 longestIncreasingSubsequence(array, solutions, current_index+1);
30 }
31
32 ArrayList<HtWt> longestIncreasingSubsequence(
33 ArrayList<HtWt> array) {
34     ArrayList<HtWt>[] solutions = new ArrayList[array.size()];
35     longestIncreasingSubsequence(array, solutions, 0);
36
37     ArrayList<HtWt> best_sequence = null;
38     for (int i = 1; i < array.size(); i++) {
39         best_sequence = seqWithMaxLength(best_sequence, solutions[i]);
40     }
41
42     return best_sequence;
43 }
44
45 /* Возвращает самую длинную последовательность */
46 ArrayList<HtWt> seqWithMaxLength(ArrayList<HtWt> seq1,
47 ArrayList<HtWt> seq2) {
48     if (seq1 == null) return seq2;
49     if (seq2 == null) return seq1;
50     return seq1.size() > seq2.size() ? seq1 : seq2;
51 }
52
53 public class HtWt implements Comparable {
54     /* объявления и т. д. */
55
56     /* используется для метода сортировки */
57     public int compareTo( Object s ) {
58         HtWt second = (HtWt) s;
59         if (this.Ht != second.Ht) {
60             return ((Integer)this.Ht).compareTo(second.Ht);
61         } else {
62             return ((Integer)this.Wt).compareTo(second.Wt);
63         }
64     }
65
66     /* Возвращем true, если "this" находится в строке перед "other."
67     * Это возможно: this.isBefore(other) и
68     * other.isBefore(this) оба = false. Это отличается от
69     * метода compareTo, где если a < b то b > a. */
70     public boolean isBefore(HtWt other) {
71         if (this.Ht < other.Ht && this.Wt < other.Wt) return true;
72         else return false;
73     }
74 }

```

Этот алгоритм требует $O(n^2)$ времени. Алгоритм, который использует $O(n \log(n))$ времени, существует, но он значительно сложнее, и маловероятно, что вы получите его на собеседовании. Однако если вы заинтересовались данным вопросом, поищите информацию на просторах Интернета.

- 10.8.** Вы обрабатываете поток целых чисел. Периодически вам требуется найти ранг числа x (количество значений $\leq x$). Какие структуры данных и алгоритмы необходимы для поддержки этих операций? Реализуйте метод `track(int x)`, вызываемый при генерировании символа, и метод `getRankOfNumber(int x)`, возвращающий количество значений $\leq x$ (не включая x).

Решение

Относительно легкий способ решить задачу — использовать массив, содержащий элементы в отсортированном порядке. Когда появляется новый элемент, ему понадобится место, а значит, придется двигать другие элементы. Реализация `getRankOfNumber` должна быть достаточно эффективной. Нам нужно выполнить бинарный поиск n и вернуть индекс.

Такое решение не очень эффективно, когда дело доходит до вставки элементов (`track(int x)`). Нам нужна структура данных, подходящая как для хранения элементов в отсортированном порядке, так и для вставки новых элементов. Для этого подойдет бинарное дерево поиска.

Вместо того чтобы вставлять элементы в массив, можно вставлять элементы в бинарное дерево поиска. Метод `track(int x)` будет занимать $O(\log n)$ времени, где n — размер дерева (если дерево сбалансировано).

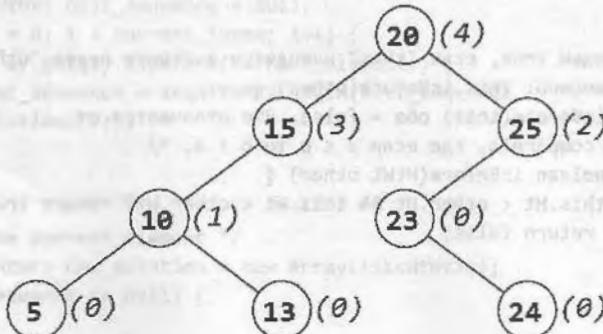
Чтобы найти позицию числа, можно выполнить симметричный обход, сохраняя при этом значение счетчика. К тому моменту, когда мы найдем x , счетчик сохранит информацию о количестве элементов, меньших, чем x .

При движении влево во время поиска x счетчик (`counter`) не меняется. Почему? Потому что мы пропускаем все значения, большие x . Наименьший элемент (с рангом 1) является крайним левым узлом.

Когда мы двигаемся вправо, то проходим все левые элементы. Они меньше чем x , поэтому нам нужно инкрементировать счетчик (`counter`) элементов в левом поддереве.

Вместо того чтобы подсчитывать размер левого поддерева (это будет неэффективно), можно отслеживать информацию при добавлении новых элементов в дерево.

Давайте рассмотрим конкретное дерево. Значения в скобках описывают количество узлов в левом поддереве (ранг узла относительно его поддерева).



Предположим, что мы хотим найти ранг значения 24. 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, это даст нам информацию, что значение 24 должно находиться левее. Значение counter не обновляется (5).

Теперь мы сравниваем 24 и 23 и понимаем, что значение 24 должно располагаться справа. Счетчик увеличивается на 1 (counter=6), поскольку 23 не имеет «левых» узлов.

Затем мы находим значение 24 и возвращаем значение счетчика counter=6.

Рекурсивный алгоритм имеет вид:

```

1 int getRank(Node node, int x) {
2     if x is node.data
3         return node.leftSize()
4     if x is on left of node
5         return getRank(node.left, x)
6     if x is on right of node
7         return node.leftSize() + 1 + getRank(node.right, x)
8 }
```

Полный код представлен ниже:

```

1 public class Question {
2     private static RankNode root = null;
3
4     public static void track(int number) {
5         if (root == null) {
6             root = new RankNode(number);
7         } else {
8             root.insert(number);
9         }
10    }
11
12    public static int getRankOfNumber(int number) {
13        return root.getRank(number);
14    }
15
16    ...
17 }
18
19 public class RankNode {
20     public int left_size = 0;
21     public RankNode left, right;
22     public int data = 0;
23     public RankNode(int d) {
24         data = d;
25     }
26
27     public void insert(int d) {
28         if (d <= data) {
29             if (left != null) left.insert(d);
30         }
31     }
32
33     public int getRank(int d) {
34         if (d <= data) {
35             return left_size + 1 + getRank(d);
36         }
37         if (right != null) return right.getRank(d);
38         return 0;
39     }
40 }
```

продолжение ➔

```

30         else left = new RankNode(d);
31         left_size++;
32     } else {
33         if (right != null) right.insert(d);
34         else right = new RankNode(d);
35     }
36 }
37
38 public int getRank(int d) {
39     if (d == data) {
40         return left_size;
41     } else if (d < data) {
42         if (left == null) return -1;
43         else return left.getRank(d);
44     } else {
45         int right_rank = right == null ? -1 : right.getRank(d);
46         if (right_rank == -1) return -1;
47         else return left_size + 1 + right_rank;
48     }
49 }
50 }

```

Обратите внимание, что мы предусмотрели случай, когда d не найден. При возникновении подобной ситуации возвращается значение -1 . Очень важно, чтобы вы на собеседовании не забыли про подобные коллизии.

Чтобы найти позицию числа, можно использовать рекурсию. Для этого мы будем возвращать размер дерева (если дерево балансировано).

Чтобы найти позицию числа, можно использовать рекурсию. Для этого мы будем возвращать это значение счетчика. К тому моменту, когда мы найдем d , скопируем необходимую информацию о количестве элементов, меньших d .

При движении влево во время поиска к счетчикам (counters) не меняться. Потому что мы пропускаем все значения, большие d . Наи меньший элемент (с рантом) является крайним левым узлом («левыйэк»). Поэтому счетчик для этого узла будет -1 .

Когда мы двигаемся вправо, то приходим к следующему узлу, который имеет счетчик -1 . Поэтому нам нужно обновить счетчик (counter) элементов в дереве. Помимо этого этого подсчитывается размер левого поддерева (это будет полезно, если можно отложиться на информацию при добавлении новых элементов в дерево).

Давайте рассмотрим конкретное дерево. Знаки в скобках указывают количество узлов в левом поддереве (при этом относительно каждого узла в дереве).



КОНЦЕПЦИИ И АЛГОРИТМЫ: РЕШЕНИЯ

11. Масштабируемость и ограничения памяти

11.1. 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 службы и обоснуйте свой подход. Вы можете использовать любые технологии и любой механизм предоставления информации клиентским приложениям.

Решение

Из постановки задачи понятно, что мы должны сфокусироваться на предоставлении информации клиентам. Допустим, что существуют какие-то скрипты, которые «волшебным» образом выбирают нужную нам информацию.

Давайте начнем со списка задач, которые нам необходимо будет решить:

- Удобство для клиента — мы хотим, чтобы наша служба была простой и полезной.
- Удобство для производителя — сервис должен быть максимально прост и в реализации, и в сопровождении. Нужно учитывать не только расходы на его разработку, но и стоимость обслуживания.
- Гибкость (учет будущих потребностей) — нужно отнести к задаче как к реальному проекту, который со временем придется совершенствовать, дорабатывать и т. д.
- Масштабируемость и эффективность — нельзя забывать об эффективности нашего решения, чтобы служба не испытывала избыточных нагрузок.

Учитывая все это, можно сформулировать несколько предложений.

Предложение 1

Данные могут храниться в обычных текстовых файлах, которые клиенты смогут получать через своего рода FTP-сервер. Такую систему достаточно легко обслуживать, так как файлы можно просмотреть, сделать резервную копию, но это приводит к сложностям при реализации парсинга. К тому же, если к нашему текстовому файлу будут добавлены данные, то они могут повредить механизм парсинга клиентов.

Предложение 2

Можем использовать стандартную базу данных SQL. Это предоставляет нам следующие преимущества:

- Облегчает обработку запросов, особенно в случаях, когда нужно поддерживать дополнительные функции. Например, мы сможем легко и эффективно выполнить

запрос вроде «выберите все акции, имеющие цену открытия больше, чем N, и цену закрытия меньше, чем M».

- Облегчает откат назад, резервное копирование и обеспечивает безопасность — все это относится к стандартным функциям баз данных, нам не придется «изобретать велосипед».
- Облегчает интеграцию с существующими приложениями. SQL-интеграция — стандартная функция во многих средах разработки приложений (SDE).

А что с недостатками?

- «Стрельба из пушки по воробьям» — SQL-база данных предоставляет намного больше функций, нежели нам необходимо. Нам не понадобится сложный бэкэнд SQL, чтобы хранить «несколько» битов информации.
- Базу данных невозможно «читать» без дополнительных усилий, поэтому нам придется позаботиться о просмотре и обслуживании данных, а это увеличит расходы на реализацию проекта.
- Безопасность: хотя базы данных SQL хорошо защищены, нам нужно ограничить права клиентов до необходимого минимума. Даже если клиенты не собираются «вредить», они могут попытаться выполнить «дорогие» и неэффективные запросы, создавая лишнюю нагрузку на сервер.

Эти недостатки не означают, что мы должны отказаться от SQL. Просто мы должны иметь представление о недостатках.

Предложение 3

XML — другой отличный способ распространения информации. Наши данные имеют фиксированный формат и размер: `company_name, open, high, low, closing price`. XML может выглядеть примерно так:

```

1 <root>
2   <date value="2008-10-12">
3     <company name="foo">
4       <open>126.23</open>
5       <high>130.27</high>
6       <low>122.83</low>
7       <closingPrice>127.30</closingPrice>
8     </company>
9     <company name="bar">
10    <open>52.73</open>
11    <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 простота чтения, как для компьютеров, так и для людей. Именно поэтому XML стал стандартной моделью совместного использования и распределения данных.

- В большинстве языков программирования есть библиотека, выполняющая парсинг XML, так что клиенты не испытывают никаких затруднений.
- Всегда можно вставить новые данные в XML-файл, просто добавив дополнительные узлы. Это не помешает работе клиентского парсера.
- Так как данные хранятся в XML-файле, можно использовать существующие инструменты резервного копирования. Нам не нужно реализовывать собственный инструмент для «бэкапа».

Недостатки:

- Это решение передает клиентам всю информацию, даже если им нужен только маленький «кусочек». Таким образом, оно неэффективно.
- Выполнение любых запросов потребует парсинга всего файла.

Независимо от метода, который будет использоваться для хранения данных, мы сможем предоставить клиентам веб-сервис (например, SOAP), обеспечивающий доступ к данным. Это добавляет дополнительный «слой» к нашей работе, но позволяет гарантировать безопасность и облегчить интегрирование системы.

Но клиенты будут получать только те данные, которые мы захотим им предоставить. Чистая SQL-реализация позволяет выполнить любой запрос клиента, например выбрать акции с самой высокой ценой.

Так какой вариант использовать? Здесь нет однозначного ответа. Использование текстовых файлов, вероятно, самый неудачный выбор, но вы можете привести множество аргументов в пользу SQL- или XML-решений, с веб-сервисом или без него.

Цель подобных вопросов не получить «правильный» ответ (ибо его не существует), а увидеть, как вы разрабатываете систему и оцениваете компромиссы.

11.2. Какие структуры данных вы бы стали использовать в очень больших социальных сетях вроде Facebook или LinkedIn? Опишите, как вы будете разрабатывать алгоритм, демонстрирующий круг знакомств или связь человека с человеком (Я → Боб → Сюзи → Джейсон → Ты).

Решение

Хороший способ решить эту задачу — устраниТЬ 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.

```
1 class Person {  
2     Person[] friends;  
3     // Другая информация  
4 }
```

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? Это очень неэффективно. Два пользователя могут быть «соседями», но нам придется просмотреть миллионы узлов в их поддеревьях, прежде чем связь обнаружится.

Шаг 2. Возвращаемся к миллионам пользователей

Когда мы имеем дело с огромными сервисами LinkedIn или Facebook, то не можем хранить все данные на одном компьютере. Это означает, что простая структура данных Person не будет работать — наши друзья могут оказаться на разных компьютерах. Таким образом, нам нужно заменить списки друзей списками их ID и работать с ними следующим образом:

1. Для каждого ID 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. У обоих классов есть хэш-таблицы, обеспечивающие эффективный поиск данных.

```
1 public class Server {  
2     HashMap<Integer, Machine> machines =  
3         new HashMap<Integer, Machine>();  
4     HashMap<Integer, Integer> personToMachineMap =  
5         new HashMap<Integer, Integer>();  
6  
7     public Machine getMachineWithId(int machineID) {  
8         return machines.get(machineID);  
9     }  
10  
11    public int getMachineIDForUser(int personID) {  
12        Integer machineID = personToMachineMap.get(personID);  
13        return machineID == null ? -1 : machineID;  
14    }  
15  
16    public Person getPersonWithID(int personID) {  
17        Integer machineID = personToMachineMap.get(personID);  
18        if (machineID == null) return null;  
19  
20        Machine machine = getMachineWithId(machineID);  
21        if (machine == null) return null;  
22  
23        return machine.getPersonWithID(personID);  
24    }  
25 }  
26  
27 public class Person {  
28     private ArrayList<Integer> friendIDs;  
29     private int personID;  
30  
31     public Person(int id) { this.personID = id; }  
32 }
```

```
33     public int getID() { return personID; }
34     public void addFriend(int id) { friends.add(id); }
35 }
36
37 public class Machine {
38     public HashMap<Integer, Person> persons =
39         new HashMap<Integer, Person>();
40     public int machineID;
41
42     public Person getPersonWithID(int personID) {
43         return persons.get(personID);
44     }
45 }
```

Существует несколько направлений оптимизации и дополнительные вопросы, которые следует обсудить.

Оптимизация: сократите количество переходов между компьютерами

«Путешествие» с одной машины на другую — дорогая операция (с точки зрения системных ресурсов). Вместо перехода с машины на машину в произвольном порядке работайте в пакетном режиме. Например, если пять друзей «живут» на одной машине, сначала получите информацию о них.

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 использовать информацию о стране, городе, состоянии и т. д. Это сократит количество переходов между машинами.

Вопрос: при поиске в ширину необходимо помечать посещенные узлы. Как это сделать?

При поиске в ширину мы устанавливаем флаг `visited` для посещенных узлов и храним его в классе узла. В нашем случае так поступать нельзя. Поскольку одновременно выполняется множество запросов, данный подход помешает редактировать данные. Вместо этого можно имитировать маркировку узлов с помощью хэш-таблицы, в которой будет храниться `id` узла и отметка, посещен он или нет.

Другие наущенные вопросы:

- В реальном мире происходят сбои серверов. Как это повлияет на проект?
- Как можно использовать кэширование?
- Вы производите поиск до конца графа? (Граф может быть бесконечным.) 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, чтобы выбрать место, где начинать обход графа?

Это всего лишь некоторые из множества вопросов, которые могут задать на собеседовании.

- 11.3.** Дан входной файл, содержащий четыре миллиарда целых чисел. Создайте алгоритм, генерирующий целое число, отсутствующее в файле. У вас есть 1 Гбайт памяти для этой задачи.

Дополнительно: а что если у вас всего 10 Мбайт?

Решение

В нашем распоряжении 2^{32} (или 4 миллиарда) целых чисел. У нас есть 1 Гбайт памяти, или 8 млрд бит.

8 млрд бит — вполне достаточный объем, чтобы отобразить все целые числа. Что нужно сделать?

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). Каждую переменную типа `int` можно рассматривать как 32 бита, или булевых значения.
2. Инициализировать битовый вектор нулями.
3. Просканировать все числа (`num`) из файла и вызывать `BV.set(num, 1)`.
4. Еще раз просканировать битовый вектор, начиная с индекса 0.
5. Вернуть индекс первого элемента со значением 0.

Следующий код реализует наш алгоритм:

```

1 byte[] bitfield = new byte [0xFFFFFFFF/8];
2 void findOpenNumber2() throws FileNotFoundException {
3     Scanner in = new Scanner(new FileReader("file.txt"));
4     while (in.hasNextInt()) {
5         int n = in.nextInt();
6         /* Находим соответствующее число в bitfield, используя
7          * оператор OR для установки n-го бита байта
8          * (то есть 10 будет соответствовать 2-му биту индекса 2
9          * в массиве байтов). */
10        bitfield [n / 8] |= 1 << (n % 8);
11    }
12
13    for (int i = 0; i < bitfield.length; i++) {
14        for (int j = 0; j < 8; j++) {
15            /* Получает отдельные биты каждого байта. Когда будет найден
16            * бит 0, находим соответствующее значение. */
17            if ((bitfield[i] & (1 << j)) == 0) {
18                System.out.println (i * 8 + j);
19                return;
20            }
21        }
22    }
23 }
```

Дополнительно: а что если у вас всего 10 Мбайт?

Можно найти отсутствующее число, воспользовавшись двойным проходом по данным. Давайте разделим целые числа на блоки некоторого размера (мы еще обсудим, как правильно выбрать размер). Пока предположим, что мы используем блоки

размером 1000 чисел. Так, `block0` соответствует числам от 0 до 999, `block1` – 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 идеей битового вектора, рассмотренного в первой части задачи. Нам ведь не нужны числа, не входящие в конкретный диапазон.

Как же выбрать размер блока? Давайте введем несколько переменных:

- Пусть `rangeSize` – размер диапазонов каждого блока на первом проходе.
- Пусть `arraySize` – число блоков при первом проходе. Обратите внимание, что $\text{arraySize} = 2^{32}/\text{rangeSize}$.

Нам нужно выбрать значение `rangeSize` 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 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.

Второй проход: битовый вектор

$$\begin{aligned}\text{arraySize} &= \frac{2^{32}}{\text{rangeSize}} \leq 2^{21} \\ \text{rangeSize} &\geq \frac{2^{32}}{2^{21}} \\ \text{rangeSize} &\geq 2^{11}\end{aligned}$$

Нам нужно место, чтобы хранить `rangeSize` бит. Поскольку в память помещается 2^{23} байт, мы сможем поместить 2^{26} бит в памяти. Таким образом:

$$2^{11} \leq \text{rangeSize} \leq 2^{26}$$

Мы получаем достаточно пространства для «маневра», но чем ближе к середине, которую мы выбираем, тем меньше памяти будет использоваться в любой момент времени.

Нижеприведенный код предоставляет одну реализацию для этого алгоритма:

```

1 int bitsize = 1048576; // 2^20 bits (2^17 bytes)
2 int blockNum = 4096; // 2^12
3 byte[] bitfield = new byte[bitsize/8];
4 int[] blocks = new int[blockNum];
5
6 void findOpenNumber() throws FileNotFoundException {
7     int starting = -1;
8     Scanner in = new Scanner(new FileReader ("file.txt"));
9     while (in.hasNextInt()) {
```

продолжение ↗

```

10     int n = in.nextInt();
11     blocks[n / (bitfield.length * 8)]++;
12 }
13
14 for (int i = 0; i < blocks.length; i++) {
15     if (blocks[i] < bitfield.length * 8) {
16         /* если значение < 2^20, то отсутствует как минимум 1 число
17          * в этой секции. */
18         starting = i * bitfield.length * 8;
19         break;
20     }
21 }
22
23 in = new Scanner(new FileReader("input_file.txt"));
24 while (in.hasNextInt()) {
25     int n = in.nextInt();
26     /* Если число внутри блока, в котором отсутствуют числа,
27      * мы записываем его */
28     if (n >= starting && n < starting + bitfield.length * 8) {
29         bitfield [(n-starting) / 8] |= 1 << ((n - starting) % 8);
30     }
31 }
32
33 for (int i = 0 ; i < bitfield.length; i++) {
34     for (int j = 0; j < 8; j++) {
35         /* Получаем отдельные биты каждого байта. Когда бит 0
36          * найден, находим соответствующее значение. */
37         if (((bitfield[i] & (1 << j)) == 0) {
38             System.out.println(i * 8 + j + starting);
39             return;
40         }
41     }
42 }
43 }

```

Что если интервьюер попросит решить задачу, используя более серьезные ограничения на использование памяти? В этом случае придется сделать несколько проходов. Сначала пройдитесь по «миллионным» блокам, потом по тысячным. Наконец, на третьем проходе можно будет использовать битовый вектор.

- 11.4.** Дано: массив целых чисел от 1 до N, где N не превышает 32 000. Значения в массиве могут повторяться, значение N неизвестно. Выведите список всех повторяющихся элементов массива, имея в распоряжении 4 Кбайт оперативной 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 пройтись по массиву, отмечая каждый элемент v , устанавливая его значение в 1. Когда мы обнаружим повторяющееся значение, то выведем его.

```
1 public static 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; // битовый набор 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]; // делим на 32  
19     }  
20  
21     boolean get(int pos) {  
22         int wordNumber = (pos >> 5); // делим на 32  
23         int bitNumber = (pos & 0x1F); // mod 32  
24         return (bitset[wordNumber] & (1 << bitNumber)) != 0;  
25     }  
26  
27     void set(int pos) {  
28         int wordNumber = (pos >> 5); // делим на 32  
29         int bitNumber = (pos & 0x1F); // mod 32  
30         bitset[wordNumber] |= 1 << bitNumber;  
31     }  
32 }
```

Обратите внимание, что задача не очень сложная, значит, решение должно быть максимально корректным. Именно поэтому мы реализовали собственный класс, чтобы хранить большой битовый вектор. Если интервьюер разрешил, можно использовать встроенный в Java класс `BitSet`.

11.5. Как избежать зацокливаний при разработке поискового робота?

Решение

Прежде всего, давайте зададим 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 хэш-таблицы, в которой после посещения страницы v устанавливается $\text{hash}[v] = \text{true}$.

Подобное решение применимо при поиске в ширину. Каждый раз при посещении страницы мы собираем все ее ссылки и добавляем их в конец очереди. Если мы уже посетили страницу, то просто ее игнорируем.

Здорово, но что означает посетить страницу v ? Что определяет страницу v : ее содержимое или URL?

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, а страница от этого не изменится. Например, страница www.careercup.com?foobar=hello — это та же страница, что и www.careercup.com.

Вы можете сказать: «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 не существует идеального способа идентифицировать страницу, и задача превращается в головоломку.

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 можно создать своего рода подпись, основанную на фрагментах контента и URL-адресе.

Давайте посмотрим, как такой алгоритм может работать.

Допустим, что существует база данных, хранящая список элементов, которые необходимо проверить. При каждой итерации мы выбираем страницу с самым высоким приоритетом:

1. Открываем страницу и создаем подпись страницы, основанную на определенных подсекциях страницы и ее URL.
2. Запрашиваем базу данных, чтобы увидеть, когда посещалась страница с этой подписью.
3. Если элемент с такой подписью недавно проверялся, то присваиваем низший приоритет и возвращаем страницу в базу данных.
4. Если элемент новый, то совершаем обход страницы и добавляем ее ссылки в базу данных.

Такой алгоритм не позволит нам полностью обойти Всемирную паутину, но предотвратит зацикливание. Если нам понадобится возможность полного обхода страницы (подходит для небольших интранет-систем), можно просто понижать приоритет так, чтобы страница все равно проверялась.

Это упрощенное решение, но есть множество других, которые тоже можно использовать. Обсудите с интервьюером нюансы, чтобы понять, каким способом лучше решить задачу. Фактически, обсуждение этой задачи может трансформироваться в задачу 11.6.

11.6. Дано: 10 миллиардов URL-адресов. Как обнаружить все дублирующиеся документы? Дубликатом считается совпадение URL-адреса.

Решение

Сколько пространства понадобится для хранения 10 миллиардов URL-адресов? Если в среднем URL-адрес занимает 100 символов, а каждый символ представляется 4 байтами, то для хранения списка из 10 миллиардов URL понадобится около 4 Тбайт. Скорее всего, нам не понадобится хранить так много информации в памяти. Давайте попробуем сначала решить упрощенную версию задачи. Представим, что в памяти хранится весь список URL. В этом случае можно создать хэш-таблицу, где каждому дублирующемуся URL ставится в соответствие значение `true` (альтернативное решение: можно просто отсортировать список и найти дубликаты, это займет некоторое время, но даст и некоторые преимущества).

Теперь, когда у нас есть решение упрощенной версии задачи, можно перейти к 400 Гбайт данных, которые нельзя хранить в памяти полностью. Давайте сохраним некоторую часть данных на диске или разделим данные между компьютерами.

Решение 1: хранение данных на диске

Если мы собираемся хранить все данные на одной машине, то нам понадобится двойной проход документа. На первом проходе мы разделим список на 400 фрагментов по 1 Гбайт в каждом. Простой способ — хранить все URL-адреса и в файле `<x>.txt`, где $x = \text{hash}(u) \% 400$. Таким образом, мы разбиваем URL-адреса по хэш-значениям. Все URL-адреса с одинаковым хэш-значением окажутся в одном файле.

На втором проходе можно использовать придуманное ранее решение: загрузить файл в память, создать хэш-таблицу URL-адресов и найти повторы.

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 преимущества и недостатки.

Основное преимущество заключается в том, что можно организовать параллельную работу так, чтобы все 400 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.

Оба решения хороши и оба требуют обсуждения с интервьюером.

11.7. Дано: веб-сервер самой простой поисковой системы. В системе есть 100 компьютеров, обрабатывающих поисковые запросы, которые могут генерировать запрос `processSearch(string query)` в другой кластер компьютеров, чтобы получить результат. Компьютер, отвечающий на запрос, выбирается случайным образом, вы не можете гарантировать, что одна и та же машина всегда будет получать один и тот же запрос. Метод `processSearch` очень затратный. Разработайте механизм кэширования новых запросов. Объясните, как при изменении данных будет обновляться кэш.

Решение

Перед проектированием системы нужно разобраться в формулировке задачи. Многие из деталей несколько неоднозначны, как и должно быть в подобных вопросах. Мы сделаем разумные предположения, но вы должны обсудить все эти детали с интервьюером.

Предположения

Давайте сформулируем несколько возможных предположений. Если вы будете использовать другой подход или другую структуру системы, то и предположения окажутся другими. Некоторые подходы могут быть чуть лучше других, но не существует единственного «правильного» решения.

- Обработка запроса происходит на исходной машине (кроме обращения к `processSearch`).
- Количество запросов, которые нужно кэшировать, — большое (миллионы запросов).
- Запросы между машинами выполняются быстро.
- Результат, возвращаемый запросом, — упорядоченный список URL, каждому URL соответствует 50 символов заголовка и 200 символов сводки.
- Самые популярные запросы остаются в кэше.

Еще раз напоминаю, что данные предположения не единственно допустимые. Это всего лишь один из возможных вариантов разумных предположений.

Системные требования

При разработке кэша нужно исходить из того, что мы должны поддерживать две первичные функции:

- обеспечить эффективный поиск по заданному ключу;
- вовремя заменять устаревшие данные на новые.

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 кэш не устареет.

Шаг 1. Проект кэша для одной системы

Хороший способ — начать с разработки кэша для одной машины. Как бы вы создали структуру данных, позволяющую исключать старые данные и производить эффективный поиск по заданному ключу?

- Связный список позволяет легко исключать старые данные, перемещая «свежие» элементы в начало. Можно удалять последний элемент связного списка, когда список достигнет заданного размера.
- Хэш-таблица позволяет эффективно искать данные, но удаление данных сопряжено с определенными трудностями.

Давайте объединим эти две структуры, чтобы получить максимум преимуществ:

- Мы создаем связный список, где узел перемещается вверх при каждом обращении к нему. Таким образом, в конце списка окажется самая неактуальная информация.

- Кроме того, мы будем использовать хэш-таблицу, которая ставит 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 позволяет проиллюстрировать работу алгоритма. Полную версию этого кода можно скачать с сайта автора книги. Впрочем, на собеседовании вас вряд ли попросят написать полный код для такой сложной 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    /* Перемещаем узел к началу связного списка */  
11    public void moveToFront(Node node) { ... }  
12    public void moveToFront(String query) { ... }  
13  
14    /* Удаляем узел из связного списка */  
15    public void removeFromLinkedList(Node node) { ... }  
16  
17    /* Получаем результат из кэша и обновляем связный список */  
18    public String[] getResults(String query) {  
19        if (!map.containsKey(query)) return null;  
20  
21        Node node = map.get(query);  
22        moveToFront(node); // обновляем "свежесть"  
23        return node.results;  
24    }  
25  
26    /* Вставляем результаты в связный список и хэшируем */  
27    public void insertResults(String query, String[] results) {  
28        if (map.containsKey(query)) { // обновляем значения  
29            Node node = map.get(query);  
30            node.results = results;  
31            moveToFront(node); // обновляем "свежесть"  
32            return;  
33        }  
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}
```

Шаг 2. Переходим к нескольким компьютерам

Теперь, когда мы понимаем, как решить задачу для одной машины, пора перейти к случаю с несколькими компьютерами. Вспомните условие задачи: нет никакой гарантии, что конкретный запрос будет всегда попадать на один и тот же компьютер. Первое, в чем нужно разобраться, — как кэш будет использоваться многими машинами совместно. Существует 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 «вспомнится» 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 быстро, но связь между машинами никак не учитывается. Кэширование, к сожалению, оказывается менее эффективным, так как множество повторных запросов 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. Структуры данных — связный список и хэш-таблица — будут дублироваться.

В этом случае общие запросы почти всегда попадают в кэш, 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 компьютер i нуждается в результатах запроса, он должен выяснить, какая машина содержит нужное значение, а затем попросить эту другую машину (j), чтобы она нашла запрос в кэше j .

Но как узнать, какая машина содержит нужную часть хэш-таблицы?

Один из вариантов — назначать запросы по формуле $\text{hash}(\text{query}) \% N$. Тогда компьютер i сможет использовать эту формулу, чтобы 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 возвращала пустой указатель, если в ее кэше нет этого запроса. После этого машина i сама вызывает `processSearch(query)` и отправит результаты машине j для хранения. Однако данная реализация увеличивает количество обращений от одной машине к другой.

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»

позволяющий кэшируемым результатам обновляться (периодически или «по запросу»), когда определенный контент изменится.

Давайте сначала разберемся, в каких случаях происходит изменение результатов (обсудите этот вопрос с интервьюером). Вот 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 хэш-таблицу, которая покажет, какие кэшированные запросы связаны с конкретным URL-адресом. Данную задачу можно решать отдельно от кэширования и располагать ресурсы на других компьютерах. 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.

Ситуацию 3 существенно труднее обработать. Можно обновлять запросы из одного слова, выполняя парсинг содержимого нового URL и удаляя однословные запросы из кэшей. Но в этом случае мы обрабатываем только однословные запросы.

Хороший способ обработать ситуацию 3 — предусмотреть «автоматический таймаут» для кэша. Таким образом, независимо от того, насколько запрос популярен, он не сможет находиться в кэше больше x минут. Это гарантирует, что все данные будут периодически обновляться.

Шаг 4. Перспективы

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. Возьмём «экстремальный» пример — 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 присваивать запросы машинам на основании хэш-значения, а не в произвольном порядке. Такое решение обладает и достоинствами, и недостатками.

Еще одна оптимизация, которую мы можем реализовать, — автоматический таймаут. 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). Нужно учитывать, как часто страница обновлялась до этого. Таймаут для каждого URL должен быть минимальным.

Это только несколько оптимизаций, которые можно сделать. Не существует единственного правильного способа решить подобную задачу. Такие задания должны продемонстрировать интервьюеру, как вы проектируете систему, какие подходы и методы вы используете.

12. Тестирование

12.1. Найдите ошибку (ошибки) в следующем коде:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3 printf("%d\n", i);
```

Решение

В коде есть две ошибки.

Первая заключается в том, что используется тип `unsigned int`, который работает только со значениями, большими или равными нулю. Поэтому условие цикла `for` всегда будет истинно, и цикл будет выполняться бесконечно.

Корректный код, выводящий значения всех чисел от 100 до 1, должен использовать условие `i > 0`. Если нам на самом деле нужно вывести нулевое значение, то следует добавить дополнительный оператор `printf` после цикла `for`.

```
1 unsigned int i;
2 for (i = 100; i > 0; --i)
3 printf("%d\n", i);
```

Вторая ошибка — вместо `%d` следует использовать `%u`, поскольку мы выводим целые значения без знака.

```
1 unsigned int i;
2 for (i = 100; i > 0; --i)
3 printf("%u\n", i);
```

Теперь этот код правильно выведет список чисел от 100 до 1, в убывающем порядке.

12.2. У вас есть 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 С. Какие ошибки могут вызвать падение приложения? Как вы проверите каждую?

Решение

Вопрос в значительной степени зависит от типа диагностируемого приложения. Однако мы можем привести некоторые общие причины случайных отказов.

1. «Случайная» переменная: приложение может использовать некоторое «случайное» значение или переменную-компонент, которая не имеет конкретного точного значения. Примеры: ввод данных пользователем, случайное число, сгенерированное программой, время суток 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 выполняется по-разному.

3. Утечка памяти: программа, возможно, исчерпала все ресурсы. Другие причины носят случайный характер и зависят от количества запущенных в определенное время процессов. Сюда же можно отнести переполнение кучи или повреждение данных в стеке.
4. Внешние причины: программа может зависеть от другого приложения, машины или ресурса. Если таких связей много, программа может «упасть» в любой момент. Чтобы найти проблему, нужно максимально изучить приложение. Кто его запускает? Что делают пользователи? Что делает само приложение?

Хотя приложение падает не в каком-то конкретном месте, возможно, само падение связано с конкретными компонентами или сценариями. Например, приложение может оставаться работоспособным в момент запуска, а сбой происходит только после загрузки файла. Или же сбой происходит в зоне ответственности компонентов низкого уровня, например при файловом вводе-выводе.

Можно делать выборочное тестирование. Закройте все остальные приложения. Очень внимательно отслеживайте все свободные ресурсы. Если есть возможность отключить части программы, сделайте это. Запустите программу на другой машине и посмотрите, возникнет ли эта ошибка. Чем больше мы можем изменить, тем легче найти проблему. Кроме того, можно использовать специальные инструменты проверки специфических ситуаций. Например, чтобы исследовать причину появления ошибок 2-го типа, можно использовать отладчики, проверяющие неинициализированные переменные. Подобные задачи позволяют вам продемонстрировать не только 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? Хотелось бы надеяться на последнее.

12.3. Дано: игра «Шахматы». В ней реализован 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 экстремального случая (убедитесь, что программа не «рухнет» при неправильном вводе) и тестирование общего случая. Мы начнем с первого.

Тип тестирования 1: проверка экстремального случая

Нам нужно убедиться, 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 будет пустой (или почти пустой).
- Проверить, что произойдет, если белых фигур больше, чем черных.
- Проверить, что произойдет, если черных фигур больше, чем белых.

Мы должны выяснить у интервьюера, как следует поступать в перечисленных случаях: возвращать `false` или генерировать исключение (и соответственно протестировать и этот случай).

Тип тестирования 2: общее тестирование

Общее тестирование — более объемная задача. В идеале, нужно протестировать все возможные варианты расстановки фигур, но их слишком много. Значит, нам следует проанализировать все разумные варианты.

В шахматах используется 6 фигур, поэтому мы можем протестировать все варианты возможных расстановок пар фигур. Это можно сделать с помощью следующего кода:

```
1 foreach части а:
2     for each other type of piece b (6 types + empty space)
3         foreach direction d
4             Создать доску с фигурой а.
5             Поместить фигуру b в точку d.
6             Попытаться сделать ход, проверить возвращаемое значение.
```

Ключ к решению этой задачи — не нужно пытаться протестировать все возможные сценарии. Вместо этого следует сосредоточиться на существенных областях.

12.4. Как вы проведете нагрузочный тест веб-страницы без использования специальных инструментов тестирования?

Решение

Тестирование нагрузки помогает идентифицировать максимальную операционную емкость веб-приложения и выявить любые узкие места, которые могут оказать влияние на его производительность. Также мы можем проверить, как приложение реагирует на изменения нагрузки.

Чтобы провести нагрузочный тест, нам нужно сначала идентифицировать критические сценарии производительности и необходимые метрики. Типичные критерии:

- время отклика;
- пропускная способность;
- использование ресурсов;
- максимальная нагрузка, которую может выдержать система.

Затем можно приступить к разработке тестов, чтобы смоделировать нагрузку, учитывая каждый из этих критерии.

При отсутствии инструментов тестирования можно создать собственные. Например, можно смоделировать одновременную работу множества людей, создав тысячи виртуальных пользователей. Можно написать многопоточную программу с тысячами потоков, где каждый поток будет соответствовать реальному пользователю, загружающему страницу. Для каждого пользователя следует программно измерить основные параметры: время отклика, ввод-вывод данных и т. д.

Затем следует проанализировать результаты, основанные на данных, собранных во время тестов, и сравнить их с базовыми показателями.

12.5. Как вы будете тестировать авторучку?

Решение

Эта задача о понимании ограничений и структурном подходе к решению.

Чтобы понять ограничения, вы должны задать много вопросов и разобраться «кто, что, где, когда, как и почему». 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 протестирували авторучку?

Кандидат: Позвольте мне 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 одежде? На стене?

Интервьюер: На одежде.

Кандидат: Замечательно! О какой ручке идет речь? Маркер или шариковая? Должен рисунок отмываться или быть постоянным?

Интервьюер: Должен отмываться.

Много вопросов спустя вы добираетесь до сути:

Кандидат: Хорошо, я понял, что у нас есть ручка, предназначенная для детей 5–10 лет. Это маркер, позволяющий рисовать красным, зеленым, синим и черным цветом. Рисовать нужно на одежде, рисунок должен отмыться. Это правильно?

Теперь у кандидата гораздо больше данных, чем было изначально. Много интервьюеров преднамеренно дают задачу, которая кажется простой и понятной (ведь все знают, что такое ручка), чтобы дать понять, что задача может сильно отличаться от того, что вы предполагали изначально. Этот подход основан на том, что пользователи делают то же самое, но делают это случайно.

Теперь, когда вы понимаете, что тестируете, можно подумать над решением. Ключ к нему — структура. В этом случае компонентами могут быть:

- Проверка факта: удостовериться, что ручка является маркером и что используются чернила нужных цветов.
- Намеченное использование: рисование. Ручка оставляет следы на одежде?
- Намеченное использование: стирка. Чернила отстирываются? (Даже если рисунок был сделан очень давно?) Они отстирываются в горячей, теплой и холодной воде?
- 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.

Помните, что в любой задаче на тестирование вам нужно проверить использование предмета по назначению и не по назначению. Люди не всегда используют вещи так, как было задумано изначально.

12.6. Как вы будете тестировать банкомат, подключенный к распределенной банковской системе?

Решение

Первое, что нужно сделать, — добиться уточнений. 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 нам нужно проверить? У нас есть доступ к программному обеспечению или только к банкомату?

Помните: хороший тестер должен знать, что тестирует!

Как только мы понимаем, на что похожа система, то можем разбить задачу на подзадачи:

- идентификация;
- снятие средств;
- внесение средств;
- проверка баланса;
- перевод денег.

Мы, скорее всего, захотим использовать ручное и автоматизированное тестирование вместе.

Ручное тестирование позволяет проверить все возможные ошибки (низкий баланс, новый счет, несуществующий счет и т. д.).

Автоматизированное тестирование сложнее. Мы автоматически тестируем все стандартные сценарии и проверяем некоторые специфические случаи, например работу с «пиковой» нагрузкой. В идеале нужно настроить закрытую систему с поддельными счетами и убедиться, что если кто-нибудь попытается быстро снять деньги с разных счетов, то он их не получит.

Прежде всего, мы расставляем приоритеты безопасности и надежности. Счета клиентов должны быть защищены, и мы обязаны убедиться в этом. Никто не хочет неожиданно потерять свои деньги! Хороший тестер понимает, как важна система приоритетов.

13. С и С++

13.1. Напишите на С++ метод, выводящий последние K строк входного файла.

Решение

Можно действовать прямо — подсчитать количество строк (N) и вывести строки с $N-K$ до N . Для этого понадобится дважды прочитать файл, что очень неэффективно. Давайте найдем решение, которое потребует прочитать файл только один раз и выведет последние K строк.

Можно создать массив для K строк и прочитать последние K строк. В нашем массиве там будут храниться строки от 1 до K , затем от 2 до $K+1$, затем от 3 до $K+2$ и т. д. Каждый раз, считывая новую строку, мы будем удалять самую старую строку из массива.

Вы можете удивиться: 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 «закольцевать» массив.

Используя такой массив, читая новую строку, мы всегда будем заменять самый старый элемент. Самый старый элемент будет храниться в отдельной переменной, которая будет меняться при добавлении новых элементов.

Пример использования закольцованного 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
```

Приведенный далее код реализует этот алгоритм:

```
1 void printLast10Lines(char* fileName) {
2     const int K = 10;
3     ifstream file (fileName);
4     string L[K];
5     int size = 0;
6
7     /* читаем файл построчно в круговой массив */
8     while (file.good()) {
9         getline(file, L[size % K]);
10        size++;
11    }
12
13    /* вычисляем начало кругового массива и его размер */
14    int start = size > K ? (size % K) : 0;
15    int count = min(K, size);
```

```

17     /* выводим элементы в порядке чтения */
18     for (int i = 0; i < count; i++) {
19         cout << L[(start + i) % K] << endl;
20     }
21 }
```

Мы считываем весь файл, но в памяти хранится только 10 строк.

- 13.2.** Сопоставьте хэш-таблицу и *тар* из стандартной библиотеки шаблонов (STL). Как организована хэш-таблица? Какая структура данных будет оптимальной для небольших объемов данных?

Решение

В хэш-таблицу значение попадает при вызове хэш-функции с ключом. Сами значения хранятся в неотсортированном порядке. Так как хэш-таблица использует ключ для индексации элементов, вставка или поиск данных занимает $O(1)$ времени (с учетом минимального количества коллизий в хэш-таблицах). В хэш-таблице также нужно обрабатывать потенциальные коллизии. Для этого используется цепочка — связный список всех значений,ключи которых отображаются в конкретный индекс.

тар (STL) вставляет пары ключ/значение в дерево двоичного поиска, основанное на ключах. При этом не требуется обрабатывать коллизии, а так как дерево сбалансировано, время вставки и поиска составляет $O(\log N)$.

Как реализована хэш-таблица?

Хэш-таблица реализуется как массив связных списков. Когда мы хотим вставить пару ключ/значение, то, используя хеш-функцию, отображаем ключ в индекс массива. При этом значение попадает в указанную позицию связного списка.

Нельзя сказать, что элементы связного списка с определенным индексом массива имеют один и тот же ключ. Скорее, функция `hashFunction(key)` для этих значений совпадает. Поэтому, чтобы получить значение, соответствующее ключу, мы должны хранить в каждом узле и ключ и значение.

Подведем итог: хэш-таблица реализуется как массив связных списков, где каждый узел списка содержит два компонента: значение и исходный ключ. Давайте перечислим особенности реализации хэш-таблиц:

1. Нужно использовать хорошую хеш-функцию, чтобы гарантировать, что ключи были правильно распределены. Если ключи будут плохо распределены, то возникнет множество коллизий и скорость нахождения элемента снизится.
2. Независимо от того, насколько хороша наша хеш-функция, коллизии будут возникать, и мы будем нуждаться в их обработке. Это подразумевает использование цепочек связных списков (или другой метод решения проблемы).
3. Можно реализовать методы динамического увеличения или уменьшения размера хэш-таблицы. Например, когда отношение количества элементов к размеру таблицы превышает определенное значение, следует увеличить размер хэш-таблицы. Это означает, что нам потребуется создать новую хэш-таблицу и передать в нее записи из старой. Поскольку это очень трудоемкий процесс, нужно сделать все возможное, чтобы размер таблицы не менялся слишком часто.

Что может заменить хэш-таблицу при работе с небольшими объемами данных?

Можно использовать `map` (из STL) или бинарное дерево. Хотя это потребует $O(\log(n))$ времени, объем данных не велик, поэтому временные затраты будут незначительными.

13.3. Как работают виртуальные функции в C++?

Решение

Виртуальная функция определяется `vtable` (виртуальной таблицей). Если какая-либо функция класса объявлена как виртуальная, создается `vtable`, которая хранит адреса виртуальных функций этого класса. Для всех таких классов компилятор добавляет скрытую переменную `vptr`, которая указывает на `vtable`. Если виртуальная функция не переопределена в производном классе, `vtable` производного класса хранит адрес функции в родительском классе. Таблица `vtable` используется для получения доступа к адресу при вызове виртуальной функции. Механизм `vtable` позволяет реализовать динамическое связывание в C++.

Когда мы связываем объект производного класса с указателем базового класса, переменная `vptr` указывает на `vtable` производного класса. Это присвоение гарантирует, что будет вызвана нужная виртуальная функция.

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 class Triangle: public Shape {
10    public:
11        int circumference () {
12            cout << "Circumference of Triangle Class\n";
13            return 3 * edge_length;
14        }
15    };
16 void main() {
17     Shape * x = new Shape();
18     x->circumference(); // "Circumference of Base Class"
19     Shape *y = new Triangle();
20     y->circumference(); // "Circumference of Triangle Class"
21 }
```

В предыдущем примере функция `circumference` — виртуальная функция из класса `Shape`, значит, она является виртуальной в каждом из произвольных классов (`Triangle` и т. д.). В C++ разрешены вызовы невиртуальных функций во время компиляции со статическим связыванием, а вызовы виртуальной функции допускаются при динамическом связывании.

13.4. Какая разница между глубоким и поверхностным копированием? Объясните, как использовать эти виды копирования.

Решение

Поверхностное копирование копирует все значения членов класса из одного объекта в другой. Глубокое копирование делает все то же самое, но кроме этого копирует все 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 = malloc(strlen(src.ptr) + 1);
11    memcpy(dest.ptr, src.ptr);
12 }
```

Обратите внимание, что `shallow_copy` может выдавать ошибки 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 используется редко. В большинстве случаев используется глубокое копирование, особенно если размер копируемой структуры не велик.

13.5. Каково назначение ключевого слова `volatile` в C?

Решение

`volatile` информирует компилятор, что значение переменной может меняться извне. Это может произойти под управлением операционной системы, аппаратных средств или другого потока. Поскольку значение может измениться, компилятор каждый раз загружает его из памяти.

Волатильную целочисленную переменную можно объявить как:

```
int volatile x;
volatile int x;
```

Чтобы объявить указатель на эту переменную, нужно сделать следующее:

```
volatile int * x;
int volatile * x;
```

Волатильный указатель на неволатильные данные используется редко, но допустим:

```
int * volatile x;
```

Если вы хотите объявить волатильный указатель на волатильную область памяти, необходимо сделать следующее:

```
1 volatile * volatile x;
```

Волатильные переменные не оптимизированы, что может пригодиться. 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 кажется, программа зациклится. Компилятор может оптимизировать ее следующим образом:

```
1 void Fn(void) {
2     start:
3     int opt = 1;
4     if (true)
5         goto start;
6 }
```

Вот теперь цикл точно станет бесконечным. Однако внешняя операция позволит записать 0 в переменную opt и прервать цикл.

Предотвратить такую оптимизацию можно с помощью ключевого слова `volatile`, например объявить, что некий внешний элемент системы изменяет переменную:

```
1 volatile int opt = 1;
2 void Fn(void) {
3     start:
4     if (opt == 1) goto start;
5     else break;
6 }
```

Волатильные переменные используются как глобальные переменные в многопотоковых программах — любой поток может изменить общие переменные. Мы не хотим оптимизировать эти переменные.

13.6. Почему деструктор базового класса должен объявляться виртуальным?

Решение

Давайте разберемся, зачем нужны виртуальные методы. Рассмотрим 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();
```

Вызывая `p->f()`, мы обращаемся к `Foo::f()`. Это потому, что `p` — это указатель на `Foo`, а `f()` — невиртуальная функция.

Чтобы гарантировать, что `p->f()` вызовет нужную реализацию `f()`, необходимо объявить `f()` как виртуальную функцию.

Теперь вернемся к деструктору. Деструкторы предназначены для очистки памяти и ресурсов. Если деструктор `Foo` не является виртуальным, то при уничтожении объекта `Bar` все равно будет вызван деструктор базового класса `Foo`.

Поэтому деструкторы объявляют виртуальными — это гарантирует, что будет вызван деструктор для производного класса.

- 13.7.** Напишите метод, получающий указатель на структуру `Node` как параметр и возвращающий полную копию переданной структуры данных. Структура данных `Node` содержит два указателя на другие узлы (другие структуры `Nodes`).

Решение

Алгоритм будет сопоставлять адреса узла с соответствующим узлом новой структуры. Преобразование данных позволит обнаруживать узлы, которые были скопированы во время традиционного обхода структуры в глубину. При обходе посещенные узлы часто помечаются, а сама метка может принимать множество разнообразных видов, ее необязательно хранить в узле.

Таким образом, мы получаем достаточно простой рекурсивный алгоритм:

```

1  typedef map<Node*, Node*> NodeMap;
2
3  Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
4      if(cur == NULL) {
5          return NULL;
6      }
7
8      NodeMap::iterator i = nodeMap.find(cur);
9      if (i != nodeMap.end()) {
10          // мы были тут раньше, возвращаем копию
11          return i->second;
12      }
13
14      Node * node = new Node;
15      nodeMap[cur] = node; // карта перед обходом ссылок
16      node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
17      node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
18      return node;
19 }
20
21 Node * copy_structure(Node * root) {
22     NodeMap nodeMap; // нам нужна пустая карта
23     return copy_recursive(root, nodeMap);
24 }
```

- 13.8. Напишите класс для интеллектуального указателя. Это тип данных (обычно использующий шаблоны), симулирующий указатель и производящий автоматический сбор мусора. Он автоматически подсчитывает количество ссылок на объект `SmartPointer<T*>` и освобождает объект типа `T`, когда число ссылок становится равно 0.

Решение

Интеллектуальный указатель — это тот же обычный указатель, обеспечивающий безопасность благодаря автоматическому управлению памятью. Такой указатель помогает избежать множества проблем: «висячие» указатели, «утечки» памяти и отказы в выделении памяти. Интеллектуальный 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 подходов к решению — разделить задачу на две части: 1) обрисовать общий подход и создать псевдокод, а затем 2) написать подробный код. Нам нужна переменная — счетчик ссылок, которая будет увеличиваться, как только мы добавляем новую ссылку на объект, и уменьшаться, когда мы ее удаляем. Наш псевдокод может иметь следующий вид:

```
1 template <class T> class SmartPointer {
2     /* Класс интеллектуального указателя нуждается в указателях на собственно
3      * себя и на счетчик ссылок. Оба они должны быть указателями, а не реальным
4      * объектом или значением счетчика ссылок, так как цель интеллектуального
5      * указателя - в подсчете количества ссылок через множество интеллектуальных
6      * указателей на один объект. */
7     T * obj;
8     unsigned * ref_count;
9 }
```

Для этого класса нам понадобятся конструктор и деструктор, поэтому опишем их:

```
1 SmartPointer(T * object) {
2     /* Мы хотим установить значение T * obj и установить счетчик
3      * ссылок в 1. */
4 }
5
6 SmartPointer(SmarterPointer<T>& sptr) {
7     /* Этот конструктор создает новый интеллектуальный указатель на существующий
8      * объект. Нам нужно сперва установить obj и ref_count
9      * в sptr's obj и ref_count. Затем,
10     * поскольку мы создали новую ссылку на obj, нам нужно
11     * увеличить ref_count. */
12 }
13
14 ~SmartPointer(SmarterPointer<T> sptr) {
15     /* Уничтожаем ссылку на объект. Уменьшаем
16     * ref_count. Если ref_count = 0, освобождаем память и
17     * уничтожаем объект. */
18 }
```

Существует дополнительный способ создания ссылок — установка одного SmartPointer в другой. Нам понадобится переопределить оператор `equal` для обработки этого случая, но сначала давайте сделаем набросок кода:

```
19 onSetEquals(SmartPointer<T> ptr1, SmartPointer<T> ptr2) {
20     /* Если ptr1 имеет существующее значение, уменьшить его количество ссылок.
21      * Затем копируем указатели obj и ref_count. Наконец,
22      * так как мы создали новую ссылку, нам нужно увеличить
23      * ref_count. */
24 }
```

Осталось только написать код, а это — дело техники:

```
1 template <class T> class SmartPointer {
2     public:
3         SmartPointer(T * ptr) {
4             ref = ptr;
5             ref_count = (unsigned*)malloc(sizeof(unsigned));
6             *ref_count = 1;
7         }
8
9         SmartPointer(SmartPointer<T> & sptr) {
10            ref = sptr.ref;
11            ref_count = sptr.ref_count;
12            ++(*ref_count);
13        }
14
15        /* Перезаписываем оператор равенства (equal), поэтому когда вы установите
16        * один интеллектуальный указатель в другой, количество ссылок старого указателя
17        * будет уменьшено, а нового — увеличено.
18        */
19        SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
20            /* Если уже присвоено объекту, удаляем одну ссылку. */
21            if (*ref_count > 0) {
22                remove();
23            }
24            if (this != &sptr) {
25                ref = sptr.ref;
26                ref_count = sptr.ref_count;
27                ++(*ref_count);
28            }
29            return *this;
30        }
31
32        ~SmartPointer() {
33            remove(); // Удаляем одну ссылку на объект.
34        }
35
36        T getValue() {
37            return *ref;
38        }
39
40    protected:
```

```

41     void remove() {
42         --(*ref_count);
43         if (*ref_count == 0) {
44             delete ref;
45             free(ref_count);
46             ref = NULL;
47             ref_count = NULL;
48         }
49     }
50
51     T * ref;
52     unsigned * ref_count;
53 };

```

Код достаточно сложный, и, скорее всего, вам не удастся избежать ошибок.

13.9. Напишите функцию динамического распределения памяти, работающую с адресами, кратными степеням двойки.

Решение

Как правило, функция динамического распределения памяти не позволяет указывать, где именно в пределах кучи будет выделена память. Мы просто получаем указатель на выделенный блок памяти.

По условию, мы должны работать с ограничениями, запрашивая нужные объемы памяти. Предположим, что нам нужен 100-байтовый блок и мы хотим, чтобы он начинался с адреса, кратного 16. Сколько памяти необходимо выделить, чтобы гарантировать выполнение условия? Нам понадобятся 15 дополнительных байтов. 100 + 15 байтов — теперь наш адрес обеспечивает делительность на 16 для 100 байтов информации.

Код будет иметь вид:

```

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 довольно хитрая, поэтому давайте ее обсудим. Предположим, что `alignment` = 16. Мы знаем, что где-то в первых 16 байтах существует адрес памяти, кратный 16. Операция AND последних трех битов адреса памяти с `000` гарантирует нам, что это новое значение будет кратно 16.

Это решение почти идеально, за исключением одного: как освобождать память?

Мы выделили дополнительные 15 байтов и должны освободить их, когда будем освобождать «реальную» память.

Можно сделать это, сохраняя в этой «дополнительной» памяти адрес начала полного блока памяти. Сохраним эту информацию перед «округленным» блоком памяти. Конечно, это означает, что нам придется выделять больший объем дополнительной памяти, чтобы хватило места для этого указателя.

К округленному с `alignment` значению в байтах требуется дополнительно выделить `alignment - 1 + sizeof(void*)` байтов.

Приведенный далее код реализует этот подход:

```

1 void* aligned_malloc(size_t required_bytes, size_t alignment) {
2     void* p1; // исходный блок
3     void** p2; // выровненный блок
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     p2[-1] = p1;
10    return p2;
11 }
12
13 void aligned_free(void *p2) {
14     /* для однозначности будем использовать те же имена, что и aligned_malloc*/.
15     void* p1 = ((void**)p2)[-1];
16     free(p1);
17 }
```

Давайте посмотрим, как работает `aligned_free`. Метод `aligned_free` передается в `p2` (этот же `p2` у нас используется в `aligned_malloc`). Мы знаем, что значение `p1` (которое указывает на начало полного блока памяти) было сохранено перед `p2`.

При работе с `p2 void**` (или массив `void**`) достаточно посмотреть на индекс -1 , чтобы получить `p1`. А освобождая `p1`, мы освобождаем целый блок памяти.

- 13.10.** Напишите на С функцию (`my2DAlloc`), которая выделяет память для двумерного массива. Минимизируйте количество вызовов функции и убедитесь, что к памяти можно обращаться как `arr[i][j]`.

Решение

Как мы знаем, двумерный массив — это массив массивов. Так как мы используем указатели с массивами, то можем использовать двойные указатели для создания двойного массива.

Основная идея заключается в создании одномерного массива указателей. Затем для каждого элемента массива можно создать новый одномерный массив. Таким образом мы получаем двумерный массив, к которому можно получить доступ через элементы массива.

Приведенный далее код реализует этот подход:

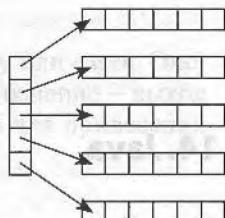
```

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 данном коде мы сообщаем `rowptr`, на что должен указывать каждый элемент. Справа представлена схема выделения памяти.

Чтобы освободить память, вызова `free` для `rowptr` будет недостаточно. Мы должны убедиться, что освободили не только память, выделенную первым вызовом `malloc`, но и память, выделенную 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 последовательный блок памяти. Для двумерного массива из 5 строк и 6 столбцов такой блок будет иметь вид:



Если использование двумерного массива кажется вам странным, вспомните, что такое решение практически не отличается от первой схемы. Единственная разница — память находится в непрерывном блоке.

Нижеприведенный код реализует это решение:

```
1 int** my2DAAlloc(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) {
7         return NULL;
8     }
9
10    int* buf = (int*) (rowptr + rows);
11    for (i = 0; i < rows; i++) {
12        rowptr[i] = buf + i * cols;
13    }
14    return rowptr;
15 }
```

Обратите внимание на строки 11–13. Если у нас 5 строк по 6 столбцов каждая, `array[0]` будет указывать на `array[5]`, `array[1]` — на `array[11]` и т. д.

Затем, когда мы вызываем `array[1][3]`, компьютер отыщет `array[1]`, который указывает на другое место в памяти — `array[5]`. Этот элемент нужно обработать как массив и получить третий элемент (нумерация с 0).

Преимущества построения массива с одним вызовом `malloc` заключается в простом освобождении памяти. Понадобится только один вызов `free`, чтобы освободить все блоки данных.

14. Java

14.1. Как повлияет на наследование объявление конструктора приватным?

Решение

Объявление конструктора приватным гарантирует, что никто за пределами класса не может создавать его экземпляры. В этом случае единственный способ получить экземпляр класса — создать статический `public`-метод, как это происходит в фабричном методе (Factory Method Pattern).

Поскольку конструктор является приватным, класс нельзя наследовать.

14.2. Будет ли выполняться блок `finally` (на Java), если оператор `return` находится внутри `try`-блока (`try-catch-finally`)?

Да, он будет выполнятся. Блок `finally` выполняется при выходе из блока `try`. Даже если мы попытаемся выйти из блока `try` принудительно (например, с помощью операторов `return`, `continue`, `break` или еще каким-либо образом), блок `finally` все равно будет выполнен.

Существует несколько случаев, когда блок `finally` не выполняется:

- виртуальная машина прекратила свою работу во время выполнения блока `try/catch`;
- поток, который выполнял блок `try/catch`, был «убит».

14.3. В чем разница между `final`, `finally` и `finalize`?

Решение

Несмотря на созвучные названия, `final`, `finally` и `finalize` имеют разные предназначения. `final` управляет «изменяемостью» переменной, метода или класса. `finally` используется в блоке `try/catch`, чтобы гарантировать, что сегмент кода будет выполнен. Метод `finalize()` вызывается сборщиком мусора, как только последний решает, что ссылок больше не существует.

Давайте поговорим об этих ключевых словах поподробнее.

`final`

Назначение оператора `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 иметь производные классы.

finally

Дополнительный блок **finally** может располагаться после блоков **try** или **catch**. Операторы, находящиеся в блоке **finally**, всегда будут выполняться (исключение — выход виртуальной машины Java из блока **try**). Блок **finally** используется для приведения кода в порядок.

finalize()

Метод **finalize()** вызывается сборщиком мусора, когда тот принимает решение, что ссылок больше нет. Обычно это используется для освобождения ресурсов, например при закрытии файла.

14.4. Объясните разницу между шаблонами в C++ и обобщениями (generic) в Java.

Решение

Многие программисты полагают, что шаблон и обобщение — это одно и то же, ведь синтаксис похож, в обоих случаях можно написать что-то вроде `List<String>`. Чтобы найти различия, давайте разберемся, *что и почему* реализуется в каждом из языков. Обобщения Java происходят от идеи «стирания типа». Эта техника устраниет параметризованные типы, когда исходный код преобразуется в JVM.

Предположим, что у вас есть 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:

```
1 Vector vector = new Vector();
2 vector.add(new String("hello"));
3 String str = (String) sv.get(0);
```

Использование обобщений Java не повлияло на наши возможности, но сделало код более красивым. Поэтому обобщения Java часто называют «синтаксическим сахаром».

Обобщения сильно отличаются от шаблонов C++. Шаблоны в C++ представляют собой набор макросов, создающих новую копию шаблона кода для каждого типа. Особенно это заметно на следующем примере: экземпляр `MyClass<Foo>` не сможет совместно с `MyClass<Bar>` использовать статическую переменную. А 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>
```

продолжение ↗

```

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 = foo1->val; // будет равно 15
23 int b1 = bar1->val; // будет равно 35
24 int b2 = bar2->val; // будет равно 35

```

В Java различные экземпляры `MyClass` могут совместно использовать статические переменные, независимо от типизированных параметров.

Из-за различий в архитектуре обобщения Java и шаблоны C++ имеют множество отличий:

- Шаблоны C++ могут использовать примитивные типы, как, например, `int`, а обобщения Java — нет, они обязаны использовать `Integer`.
- Java позволяет указывать типизированные параметры для шаблона, чтобы он был определенного типа. Например, вы можете использовать обобщения для реализации `CardDeck` и указать, что типизированный параметр должен «происходить» от `CardGame`.
- В C++ можно создать экземпляр типизированного параметра, в Java — нет.
- Java не позволяет использовать типизированные параметры (`Foo` в `MyClass<foo>`) для статических методов и переменных, так как они могут совместно использоваться `MyClass<Foo>` и `MyClass<Bar>`. В C++ — это разные классы, поэтому типизированный параметр можно использовать для статических методов и переменных.
- В Java все экземпляры `MyClass`, независимо от их типизированных параметров, относятся к одному и тому же типу. Типизированные параметры уничтожаются после выполнения. В C++ экземпляры с разными типизированными параметрами — различные типы.

Помните, что хотя обобщения Java и шаблоны C++ внешне похожи, у них много различий.

14.5. Объясните, что такое отражение объекта в Java и когда оно используется.

Решение

Рефлексия, или отражение, объекта — функция, обеспечивающая получение отражаемой 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. Непосредственное получение и установка полей объекта через ссылку поля (независимо от модификатора доступа).

Приведенный далее код демонстрирует рефлексию объекта:

```

1  /* Параметры */
2  Object[] doubleArgs = new Object[] { 4.2, 3.9 };
3
4  /* Получаем класс */
5  Class rectangleDefinition = Class.forName("MyProj.Rectangle");
6
7  /* Equivalent: 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 =
12  (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);
13
14 /* Эквивалент: область Double = rectangle.area(); */
15 Method m = rectangleDefinition.getDeclaredMethod("area");
16 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 объекта не кажется необходимой, но в некоторых случаях она очень полезна:

- Она помогает наблюдать или управлять поведением программы во время выполнения.
- Она помогает отлаживать или тестировать программы, поскольку мы получаем прямой доступ к методам, конструкторам и полям.
- Она позволяет вызывать методы по имени, даже если нам это имя заранее не было известно. Например, пользователь может передавать имя класса, параметры конструктора и имя метода. Эту информацию можно использовать при создании объекта и вызове метода. Выполнение аналогичных операций без рефлексии потребует использования цепочки сложных `if`-операторов (если вам повезет и задача окажется реализуемой).

14.6. Реализуйте класс `CircularArray` для массива, хранящего структуру данных и обеспечивающего эффективный циклический сдвиг. Класс должен использовать обобщенный тип и поддерживать итерацию через стандартный оператор `for(Obj o : circularArray)`.

Решение

Эту задачу можно разделить на две части. Сперва нам необходимо реализовать класс `CircularArray`, а затем заняться итерацией. Рассмотрим обе части по отдельности.

Реализация класса `circulararray`

Один из вариантов реализации класса `CircularArray` — циклический сдвиг элементов при вызове `rotate(int shiftRight)`. Но этот способ нельзя назвать эффективным.

Можно создать переменную экземпляра `head`, указывающую на начало массива с циклическим сдвигом. Вместо того чтобы выполнять сдвиг, мы будем увеличивать значение `head` на `shiftRight`.

Код реализует этот подход:

```

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 много тонких мест, в которых вы можете допустить ошибки:

- Мы не можем создать массив обобщенного типа. Вместо этого `items` должен иметь тип `List<T>`.
- Оператор `%` будет возвращать отрицательное значение, если мы выполним операцию `negValue % posVal`. Например, $-8 \% 3 = -2$. Это не соответствует смыслу модуля. Чтобы получить правильный положительный результат, к отрицательному значению индекса необходимо добавить `items.length`.
- Необходимо удостовериться, что необработанный индекс был преобразован в сдвинутый. Для этого в код добавлена функция `convert`, которая используется другими методами. Даже функция `rotate` использует `convert`. Это хороший пример повторного использования кода.

Теперь у нас есть код `CircularArray` и можно заняться реализацией итератора.

Реализация интерфейса Iterator

Вторая часть задачи — научить класс `CircularArray` выполнять следующую операцию:

```
1 CircularArray<String> array = ...
2 for (String s : array) { ... }
```

Для реализации этого нам потребуется создать интерфейс `Iterator`, который позволит:

- изменить определение `CircularArray<T>` и добавить `implements Iterable<T>`; добавить метод `iterator()` в `CircularArray<T>`;
- создать `CircularArrayIterator()`, реализующий `Iterator<T>`; реализовать в `CircularArrayIterator` методы `hasNext()`, `next()` и `remove()`.

Как только мы сделаем это, цикл заработает.

В приведенном далее коде мы удалили те аспекты `CircularArray`, которые идентичны предыдущей реализации:

```
1 public class CircularArray<T> implements Iterable<T> {
2     ...
3     public Iterator<T> iterator() {
4         return new CircularArrayIterator<T>(this);
5     }
6
7     private class CircularArrayIterator<TI> implements Iterator<TI>{
8         /* отражает смещение от повернутого head, не от
9          * физического начала "сырого" массива */
10        private int _current = -1;
11        private TI[] _items;
12
13        public CircularArrayIterator(CircularArray<TI> array){
14            _items = array.items;
15        }
16
17        @Override
18        public boolean hasNext() {
19            return _current < items.length - 1;
20        }
21
22        @Override
23        public TI next() {
24            _current++;
25            TI item = (TI) _items[convert(_current)];
26            return item;
27        }
28
29        @Override
30        public void remove() {
31            throw new UnsupportedOperationException("...");
32        }
33    }
34 }
```

Обратите внимание, что первая итерация цикла `for` вызывает `hasNext()`, а затем `next()`. Убедитесь, что ваша реализация будет возвращать правильные значения.

Если вам на собеседовании достанется подобная задача, возможно, вы не вспомните точные названия методов и интерфейсов. Все равно попытайтесь решить задачу как можете, тем самым вы продемонстрируете высокий уровень понимания темы.

```

1 package com.javarush.task.level.begin;
2
3 public class Solution {
4     public static void main(String[] args) {
5         List<String> list = new ArrayList<String>();
6         list.add("I");
7         list.add("Love");
8         list.add("Java");
9
10        for (String item : list) {
11            System.out.println(item);
12        }
13    }
14
15    public static void main(String[] args) {
16        List<String> list = new ArrayList<String>();
17        list.add("I");
18        list.add("Love");
19        list.add("Java");
20
21        for (String item : list) {
22            System.out.println(item);
23        }
24    }
25
26    public static void main(String[] args) {
27        List<String> list = new ArrayList<String>();
28        list.add("I");
29        list.add("Love");
30        list.add("Java");
31
32        for (String item : list) {
33            System.out.println(item);
34        }
35    }
36
37    public static void main(String[] args) {
38        List<String> list = new ArrayList<String>();
39        list.add("I");
40        list.add("Love");
41        list.add("Java");
42
43        for (String item : list) {
44            System.out.println(item);
45        }
46    }
47
48    public static void main(String[] args) {
49        List<String> list = new ArrayList<String>();
50        list.add("I");
51        list.add("Love");
52        list.add("Java");
53
54        for (String item : list) {
55            System.out.println(item);
56        }
57    }
58
59    public static void main(String[] args) {
60        List<String> list = new ArrayList<String>();
61        list.add("I");
62        list.add("Love");
63        list.add("Java");
64
65        for (String item : list) {
66            System.out.println(item);
67        }
68    }
69
70    public static void main(String[] args) {
71        List<String> list = new ArrayList<String>();
72        list.add("I");
73        list.add("Love");
74        list.add("Java");
75
76        for (String item : list) {
77            System.out.println(item);
78        }
79    }
80
81    public static void main(String[] args) {
82        List<String> list = new ArrayList<String>();
83        list.add("I");
84        list.add("Love");
85        list.add("Java");
86
87        for (String item : list) {
88            System.out.println(item);
89        }
90    }
91
92    public static void main(String[] args) {
93        List<String> list = new ArrayList<String>();
94        list.add("I");
95        list.add("Love");
96        list.add("Java");
97
98        for (String item : list) {
99            System.out.println(item);
100       }
101   }
102 }
```

В этом коде есть много тонких мест, в которых вы можете допустить ошибки.

- Мы не можем создать объекта `ArrayList` этого типа: `List<String>`. Вместо этого должны быть `String`.
- Очередной будет изъян из списка, а плоская строка `"I Love Java"` останется в списке. Это не соответствует требуемому результату. Чтобы получить правильный текстовый результат, к отрицательному значению индекса необходимо прибавить `item.length`.
- Необходимо учесть то, что необработанный `String` в браузере ведет себя как единственный. Для этого в `System.out.println(item)` нужно использовать `item + "` и `" + item` для других методов. Использование `System.out.println(item)` это корректное вторичное использование кода.

Следующий класс есть еще один способ визуально заняться исключением итератора.

15. Базы данных

Вопросы 15.1–15.3 относятся к следующей базе данных:

Apartments		Buildings		Tenants	
AptID	int	BuildingID	int	TenantID	int
UnitNumber	varchar	ComplexID	int	TenantName	varchar
BuildingID	int	BuildingName	varchar		
		Address	varchar		
Complexes		AptTenants		Requests	
ComplexID	int	TenantID	int	RequestID	int
ComplexName	varchar	AptID	int	Status	varchar
				AptID	int
				Description	varchar

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).

15.1. Напишите SQL-запрос для получения списка арендаторов, которые снимают более одной квартиры.

Решение

Для реализации этого SQL-запроса можно использовать выражения HAVING и GROUP BY, а затем для Tenants выполнить операцию INNER JOIN.

```

1 SELECT TenantName
2 FROM Tenants
3 INNER JOIN
4     (SELECT TenantID
5      FROM AptTenants
6     GROUP BY TenantID
7    HAVING count(*) > 1) C
8 ON Tenants.TenantID = C.TenantID

```

Каждый раз при использовании на собеседовании (или на практике) GROUP BY, убедитесь, что SELECT использует агрегатную функцию или что-то, что содержится в пределах выражения GROUP BY.

- 15.2.** 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. Как только у нас появится список, можно использовать таблицу `Buildings`.

```
1 SELECT BuildingName, Count
2 FROM Buildings
3 INNER 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 следует тщательно протестировать, даже если вы пишете код на бумаге. Вначале проверьте внутреннюю часть запроса, а потом — внешнюю.

- 15.3.** Дом № 11 находится на капитальном ремонте. Напишите запрос, который закрывает все запросы на квартиры в этом здании.

Решение

Запросы `UPDATE`, подобно запросам `SELECT`, могут содержать условие `WHERE`. Чтобы реализовать этот запрос, 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
4     (SELECT AptID
5      FROM Apartments
6      WHERE BuildingID = 11)
```

- 15.4.** Какие существуют типы связей? Объясните, чем они отличаются и почему определенные типы лучше подходят для конкретных ситуаций.

Решение

`JOIN` позволяет связать две таблицы. Каждая из этих таблиц должна иметь хотя бы одно поле, которое можно использовать для поиска записей в другой таблице. Тип связи определяет, какие записи попадут в результирующий набор.

Давайте возьмем две таблицы: в одной перечислены обычные напитки, а в другой — низкокалорийные. В каждой таблице есть два поля: 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, позволяющих связать Beverage и Calorie-Free Beverages.

- **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, отсутствующие поля из левой таблицы будут содержать значения **NULL**. Обратите внимание, если у нас есть две таблицы — A и B — и мы выполняем A **LEFT JOIN** B, то результат будет совпадать с 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 мы получим шесть записей.

15.5. Что такое денормализация? Поясните этот процесс.

Решение

Денормализация — это метод оптимизации базы данных, при котором избыточные данные добавляются к одной или более таблицам. Это помогает избежать трудоемких объединений в реляционной базе данных.

В обычной нормализованной базе данных данные хранятся в отдельных логических таблицах, и задача — минимизировать резервирование. Мы стремимся хранить все фрагменты данных 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 нам понадобится получить список всех курсов с именами преподавателей, необходимо будет связать эти две таблицы.

С одной стороны, это хорошо: если учитель изменит свое имя, нам понадобится обновить информацию только в одном месте.

Но если таблицы будут больными, нам придется потратить довольно много времени на объединение таблиц.

Денормализация — это компромисс. При денормализации мы решаем, что некоторые данные должны храниться с избыточностью, но за счет этого повышаем эффективность работы.

Недостатки денормализации	Преимущества денормализации
Операции обновления и вставки записей весьма ресурсоемки	Данные можно получить быстрее, так как не нужно тратить время на объединение
Запросы на обновление и вставку записей становятся сложнее	Запросы будут проще (следовательно, меньше вероятность ошибки), так как нужно анализировать меньшее количество таблиц
Данные могут стать противоречивыми. Какие данные считать «правильными»?	
Появляется избыточность данных	

В системах с масштабированием, подобных тем, которые применяются в крупнейших компаниях, всегда используются и нормализованные, и денормализованные базы данных.

15.6. Нарисуйте диаграмму связей для базы данных, в которой фигурируют компании, клиенты и сотрудники компаний.

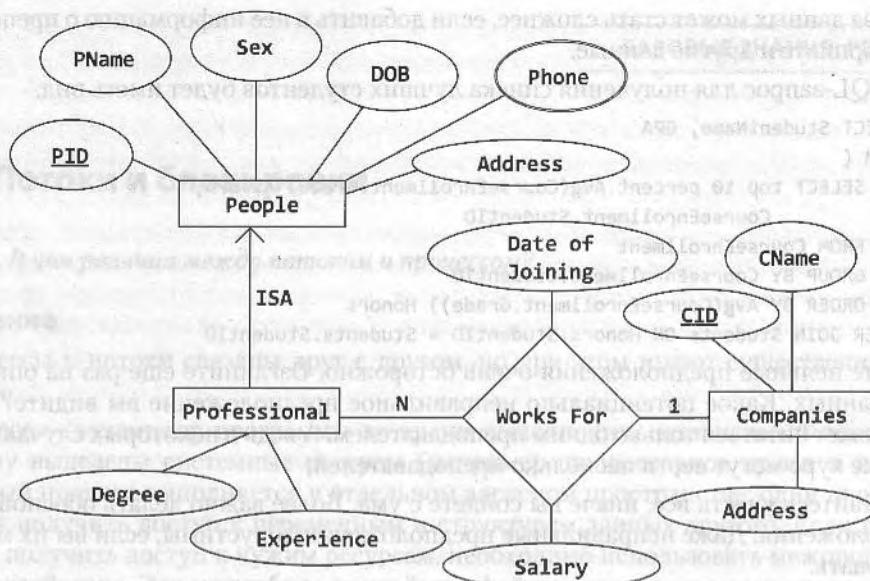
Решение

Сотрудники, работающие в компаниях (*Companies*), являются профессионалами (*Professionals*). Можно сказать, что все *Professionals* являются людьми (*People*).

У каждого профессионала существует дополнительная информация — ученая степень и опыт работы (помимо свойств, полученных от *People*).

Профессионал работает только на одну компанию, но компании (*Companies*) могут нанять много профессионалов. Таким образом, между *Professionals* и *Companies* существует связь «много-к-одному». *Works For* может хранить дополнительную информацию, например дату начала работы и зарплату. Эти атрибуты определяются при связывании *Professional* с *Company*.

У человека (*Person*) может быть несколько номеров телефона, поэтому *Phone* может содержать много значений.



15.7. Разработайте простую базу данных, содержащую данные об успеваемости студентов, и напишите запрос, возвращающий список лучших студентов (лучшие 10 %), отсортированный по их среднему баллу.

Упрощенная база данных будет содержать не меньше трех объектов: *Students* (Студенты), *Courses* (Курс лекций) и *CourseEnrollment* (Посещение курса). В *Students* хранится информация об имени студента, его ID и некоторые персональные данные. *Courses* содержит название курса, ID, описание курса, имя преподавателя и другую информацию. *CourseEnrollment* объединяет *Students* и *Courses*, а также содержит поле *CourseGrade* (оценка). Давайте считать, что *CourseGrade* — целое число.

<i>Students</i>	
<i>StudentID</i>	<i>int</i>
<i>StudentName</i>	<i>varchar(100)</i>
<i>Address</i>	<i>varchar(500)</i>

<i>Courses</i>	
<i>CourseID</i>	<i>int</i>
<i>CourseName</i>	<i>varchar(100)</i>
<i>ProfessorID</i>	<i>int</i>

<i>CourseEnrollment</i>	
<i>CourseID</i>	<i>int</i>
<i>StudentID</i>	<i>int</i>
<i>Grade</i>	<i>int</i>
<i>Term</i>	<i>int</i>

Эта база данных может стать сложнее, если добавить в нее информацию о преподавателе, зарплате и другие данные.

Наш SQL-запрос для получения списка лучших студентов будет иметь вид:

```
1 SELECT StudentName, GPA
2 FROM (
3     SELECT top 10 percent Avg(CourseEnrollment.Grade) AS GPA,
4            CourseEnrollment.StudentID
5     FROM CourseEnrollment
6    GROUP BY CourseEnrollment.StudentID
7   ORDER BY Avg(CourseEnrollment.Grade)) Honors
8 INNER JOIN Students ON Honors.StudentID = Students.StudentID
```

Делайте неявные предположения очень осторожно. Взгляните еще раз на описание базы данных. Какое потенциально неправильное предположение вы видите? Один курс может читаться только одним преподавателем. А ведь в некоторых случаях один и тот же курс могут вести несколько преподавателей.

Не пытайтесь учесть все, иначе вы сойдете с ума. Более важно делать обоснованные предположения. Даже неправильные предположения допустимы, если вы их можете обосновать.

Помните, что вам нужно найти компромисс между гибкостью и сложностью. Создание системы, учитывающей, что курс могут читать несколько преподавателей, повышает гибкость базы данных, но и увеличивает сложность. Если попытаться сделать БД, учитывающую все возможные ситуации, то в итоге мы получим нечто слишком сложное.

Сделайте проект в меру гибким и обоснуйте любые предположения или ограничения. Это касается не только проектирования баз данных, но и объектно-ориентированного проектирования и программирования.

15.6. Начало проектирования баз данных

Вопрос	Ответ
Решение	(651)-загружено
Сотрудники, работающие в компании (Company), являются профессионалами (Professionals). Можно сказать, что все Professionals являются специалистами (Profes).	(652)-загружено
У каждого профессионала существует дополнительная информация – уровень и опыт работы (Experience), полученная от Professionals.	изменена
Профессиональная работа ведется на的基础上 компании, подчиняющейся (Company). У каждого профессионала есть одна компания, между Professionals и Company существует связь «один к одному». Для этого можно использовать одинаковую ссылку на таблицу Company в таблице Professionals.	загружено
У человека (Person) можно есть не более одного места жительства. Необходимо поддерживать уникальный идентификатор для каждого человека.	загружено
Имя (Name) должно быть не более 50 символов.	загружено
Поддерживать уникальный идентификатор для каждого места жительства.	загружено

16. Потоки и блокировки

16.1. В чем разница между потоком и процессом?

Решение

Процессы и потоки связаны друг с другом, но при этом имеют существенные различия.

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены 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 хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать.

Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

16.2. Как оценить время, затрачиваемое на контекстное переключение?

Решение

Это довольно хитрый вопрос, но давайте рассмотрим одно из возможных решений. Контекстное переключение — это время, потраченное на переключение между двумя процессами (то есть перевод одного процесса из состояния ожидания в состояние выполнения и перевод другого процесса из состояния выполнения в состояние ожидания/завершения). Такие процессы возникают в многозадачных системах. Операционная система загружает информацию о состоянии ожидающих процессов в память и сохраняет информацию о состоянии работающего процесса.

Чтобы решить эту задачу, нам понадобятся отсечки (метки) времени для последней и первой инструкций заменяемых процессов. Время контекстного переключения — это разница между временными метками двух процессов.

Давайте рассмотрим простой пример. Предположим, что существует всего два процесса: P_1 и P_2 . P_1 выполняется, а P_2 — ожидает своей очереди. В некоторый момент времени операционная система должна поменять местами P_1 и P_2 . Давайте предположим,

что этот момент происходит на n -й инструкции процесса P_1 . Если $t_{x,k}$ — метка времени (в микросекундах) k -й инструкции процесса x , то контекстное переключение займет $t_{2,1} - t_{1,n}$ микросекунд.

Теперь самое тонкое место: как мы узнаем, что произошло переключение процесса? Конечно же, мы не можем хранить временную метку для каждой инструкции процесса.

Другая проблема — споппингом (переключением процессов) управляет алгоритм планирования операционной системы, и существует множество потоков на уровне ядра, которые также выполняют контекстные переключения. Множество процессов могут бороться за процессор или прерывания. У пользователя нет возможности контролировать эти «посторонние» контекстные переключения. Если в момент времени t_1 ядро решит обработать прерывание, то время, потраченное на контекстное переключение, увеличится.

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 играть в пинг-понг с маркером данных.

Давайте разрешим P_1 стать инициатором: он будет отправлять данные, а P_2 — получать. P_2 спит в ожидании данных от P_1 . Когда P_1 запускается, он 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.

Контекстное переключение заканчивается, и планировщик задач должен выбрать другой процесс. Так как P_2 подготовлен к работе, он автоматически будет выбран планировщиком для выполнения.

Когда P_2 работает, 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 нашей игры в пинг-понг:

1. Блокируем P_2 : он ожидает данные 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 порождает контекстное переключение.
5. P_2 запланирован (активируется) и получает маркер.
6. P_2 отправляет ответный маркер процессу P_1 .
7. P_2 пытается прочитать ответный маркер от P_1 . Это порождает контекстное переключение.
8. P_1 активируется и получает маркер.
9. P_1 отмечает время завершения.

Ключ к решению этой задачи в том, что отправка маркера данных порождает контекстное переключение. 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:

$$T = 2 \times (T_d + T_c + T_r)$$

Данную формулу легко можно вывести из следующих событий: P_1 отправляет токен (3), процессор выполняет контекстное переключение (4), P_2 получает маркер (5). Затем P_2 отправляет ответный маркер (6), процессор опять выполняет контекстное переключение (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. Это не будет контекстным переключением, так как P_1 в данный момент — рабочий процесс, и он не будет заблокирован при отправлении маркера.

Понадобится много итераций, чтобы подсчитать среднее время, потраченное на выполнение шагов 2–9: могут возникнуть неожиданные прерывания ядра или появляться дополнительные потоки, борющиеся за процессорное время. В качестве окончательного ответа нам нужно выбрать минимальное зарегистрированное значение времени.

Можно сказать, что полученный результат — это приближенное значение, зависящее от конкретной системы. Мы делаем предположение, что P_2 будет выбран для запуска, как только маркер данных становится доступным. Однако все зависит от планировщика задач, и мы не можем дать гарантий, что события будут развиваться именно так.

Ничего страшного, главное — не забудьте на собеседовании отметить, что ваше решение не совершенно.

- 16.3.** В знаменитой задаче об обедающих философах каждый из них имеет только одну палочку для еды. Философию 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 моделирование задачи, предотвращающее взаимные блокировки.

Решение

Давайте сначала реализуем упрощенную модель обеденной задачи, без учета взаимных блокировок. Нам понадобятся `Philosopher` (философ), происходящий от `Thread`, и `Chopstick` (палочка для еды): когда она поднимается, мы вызываем `lock.lock()`, а когда опускается — `lock.unlock()`.

```

1 public class Chopstick {
2     private Lock lock;
3
4     public Chopstick() {
5         lock = new ReentrantLock();
6     }
7
8     public void pickUp() {
9         lock.lock();
10    }
11 }
```

продолжение ⤵

```

12     public void putDown() {
13         lock.unlock();
14     }
15 }
16
17 public class Philosopher extends Thread {
18     private int bites = 10;
19     private Chopstick left;
20     private Chopstick right;
21
22     public Philosopher(Chopstick left, Chopstick right) {
23         this.left = left;
24         this.right = right;
25     }
26
27     public void eat() {
28         pickUp();
29         chew();
30         putDown();
31     }
32
33     public void pickUp() {
34         left.pickUp();
35         right.pickUp();
36     }
37
38     public void chew() { }
39
40     public void putDown() {
41         left.putDown();
42         right.putDown();
43     }
44
45     public void run() {
46         for (int i = 0; i < bites; i++) {
47             eat();
48         }
49     }
50 }

```

Выполнение данного кода может привести к взаимной блокировке, если у всех философов окажется левая палочка и они будут ждать правую.

Чтобы предотвратить блокировки, можно реализовать следующую стратегию — опустить левую палочку, если нет возможности получить правую.

```

1 public class Chopstick {
2     /* такой же, как и раньше */
3
4     public boolean pickUp() {
5         return lock.tryLock();
6     }
7 }

```

```

8
9 public class Philosopher extends Thread {
10     /* такой же, как и раньше */
11
12     public void eat() {
13         if (pickUp()) {
14             chew();
15             putDown();
16         }
17     }
18
19     public boolean pickUp() {
20         /* попытка поднять палочку */
21         if (!left.pickUp())
22             return false;
23
24         if (!right.pickUp()) {
25             left.putDown();
26             return false;
27         }
28         return true;
29     }
30 }

```

В приведенном коде нужно убедиться, что левая палочка опущена, если нет возможности поднять правую, и не вызывать `putDown()`, если у нас нет палочек.

16.4. Разработайте класс, обеспечивающий блокировку, так чтобы предотвратить возникновение мертвых блокировок.

Решение

Существует несколько общих способов предотвратить мертвые блокировки. Один из самых популярных — обязать процесс явно объявлять, в какой блокировке он нуждается. Тогда мы можем проверить, будет ли созданная блокировка мертвой, и если так, можно прекратить работу.

Давайте разберемся, как обнаружить мертвую блокировку. Предположим, что мы запрашиваем следующий порядок блокировок:

```

A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}

```

Это приведет к мертвой блокировке, потому что:

```

A блокирует 2, ждет 3
B блокирует 3, ждет 5
C блокирует 5, ждет 2

```

Можно представить этот сценарий в виде графа, где 2 соединено с 3, а 3 соединено с 5, а 5 соединено с 2. Мертвая блокировка описывается циклом. 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.

Этот класс будет нуждаться в методе `declare`, который использует потоки и процессы для объявления порядка, в котором будут запрашиваться ресурсы. Метод `declare` будет проверять порядок объявления, добавляя каждую непрерывную пару элементов (v, w) к графу. Впоследствии он проверит, не появилось ли циклов. Если возник цикл, он удалит добавленное ребро из графика и выйдет.

Нам нужно обсудить только один нюанс. Как мы обнаружим цикл? Мы можем обнаружить цикл с помощью поиска в глубину через каждый связанный элемент (то есть через каждый компонент графа). Существуют сложные компоненты, позволяющие выбрать все соединенные компоненты графа, но наша задача не настолько сложна.

Мы знаем, что если возникает петля, то виновато одно из ребер. Таким образом, если поиск в глубину затрагивает эти ребра, мы обнаружим петлю.

Псевдокод для этого обнаружения петли примерно следующий:

```

1 boolean checkForCycle(locks[] locks) {
2     touchedNodes = hash table(lock -> boolean)
3     инициализировать touchedNodes, установив в false каждый lock в 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. полный код ниже)
20     }
21 }
```

В данном коде можно сделать несколько поисков в глубину, но `touchedNodes` нужно инициализировать только один раз. Мы выполняем итерации, пока все значения в `touchedNodes` равны `false`.

Приведенный далее код более подробен. Для простоты мы предполагаем, что все блокировки и процессы (владельцы) последовательно упорядочены.

```

1 public class LockFactory {
2     private static LockFactory instance;
3
4     private int numberOfLocks = 5; /* по умолчанию */
5     private LockNode[] locks;
6
7     /* Отображаем процесс (владельца) в порядок,
8      * в котором владелец требовал блокировку */
```

```
9 12 private Hashtable<Integer, LinkedList<LockNode>> lockOrder;
10 13     }
11 13 private LockFactory(int count) { ... }
12 14 public static LockFactory getInstance() { return instance; }
13 15     также не блокируется от блокировок иных нитей
14 16 public static synchronized LockFactory initialize(int count) {
15 17     if (instance == null) instance = new LockFactory(count);
16 18     return instance;
17 19 }
18 20     }
19 21 public boolean hasCycle(
20 22     Hashtable<Integer, Boolean> touchedNodes,
21 23     int[] resourcesInOrder) {
22 24     /* проверяем на наличие петли */
23 25     for (int resource : resourcesInOrder) {
24 26         if (touchedNodes.get(resource) == false) {
25 27             LockNode n = locks[resource];
26 28             if (n.hasCycle(touchedNodes)) {
27 29                 return true;
28 30             }
29 31         }
30 32     }
31 33     return false;
32 34 }
33 35     }
34 36     /* Чтобы предотвратить мертвую блокировку, заставляем процессы
35 37     * объявлять, что они хотят заблокировать. Проверяем,
36 38     * что запрашиваемый порядок не вызовет мертвую блокировку
37 39     * (петлю в направленном графе) */
38 40 public boolean declare(int ownerId, int[] resourcesInOrder) {
39 41     Hashtable<Integer, Boolean> touchedNodes =
40 42         new Hashtable<Integer, Boolean>();
41 43     /* добавляем узлы в граф */
42 44     int index = 1;
43 45     touchedNodes.put(resourcesInOrder[0], false);
44 46     for (index = 1; index < resourcesInOrder.length; index++) {
45 47         LockNode prev = locks[resourcesInOrder[index - 1]];
46 48         LockNode curr = locks[resourcesInOrder[index]];
47 49         prev.joinTo(curr);
48 50     }
49 51     touchedNodes.put(resourcesInOrder[index], false);
50 52     /* если получена петля, уничтожаем этот список ресурсов
51 53     * и возвращаем false */
52 54     if (hasCycle(touchedNodes, resourcesInOrder)) {
53 55         for (int j = 1; j < resourcesInOrder.length; j++) {
54 56             LockNode p = locks[resourcesInOrder[j - 1]];
55 57             LockNode c = locks[resourcesInOrder[j]];
56 58             p.remove(c);
57 59         }
58 60     }
59 61 }
```

продолжение ⤵

```

60         return false;
61     }
62
63     /* Петля не найдена. Сохраняем порядок, который был объявлен,
64      * так как мы можем проверить, что процесс действительно вызывает
65      * блокировку в нужном порядке */
66     LinkedList<LockNode> list = new LinkedList<LockNode>();
67     for (int i = 0; i < resourcesInOrder.length; i++) {
68         LockNode resource = locks[resourcesInOrder[i]];
69         list.add(resource);
70     }
71     lockOrder.put(ownerId, list);
72
73     return true;
74 }
75
76 /* Получаем блокировку, проверяем сначала, что процесс
77   * действительно запрашивает блокировку в объявленаом порядке*/
78 public Lock getLock(int ownerId, int resourceId) {
79     LinkedList<LockNode> list = lockOrder.get(ownerId);
80     if (list == null) return null;
81
82     LockNode head = list.getFirst();
83     if (head.getId() == resourceId) {
84         list.removeFirst();
85         return head.getLock();
86     } else
87         return null;
88 }
89 }
90
91 public class LockNode {
92     public enum VisitState { FRESH, VISITING, VISITED };
93
94     private ArrayList<LockNode> children;
95     private int lockId;
96     private Lock lock;
97     private int maxLocks;
98
99     public LockNode(int id, int max) { ... }
100
101    /* Присоединяя "this" в "node", проверяем, что мы не создадим этим
102     * петлю (цикл) */
103    public void joinTo(LockNode node) { children.add(node); }
104    public void remove(LockNode node) { children.remove(node); }
105
106    /* Проверяем на наличие цикла с помощью поиска в глубину */
107    public boolean hasCycle(
108        Hashtable<Integer, Boolean> touchedNodes) {
109        VisitState[] visited = new VisitState[maxLocks];
110        for (int i = 0; i < maxLocks; i++) {

```

```

111         visited[i] = VisitState.FRESH;
112     }
113     return hasCycle(visited, touchedNodes);
114 }
115
116 private boolean hasCycle(VisitState[] visited,
117     Hashtable<Integer, Boolean> touchedNodes) {
118     if (touchedNodes.containsKey(lockId)) { } /* Уже посетил этот узел */
119     touchedNodes.put(lockId, true); /* Маркируем его как посещенный */
120 }
121
122 if (visited[lockId] == VisitState.VISITING) { } /* Высокоэффициент */
123 /* Мы циклически возвращаемся к этому узлу, следовательно, */
124 /* мы знаем, что здесь есть цикл (петля) */
125 return true; /* Цикл обнаружен */
126 } else if (visited[lockId] == VisitState.FRESH) { }
127     visited[lockId] = VisitState.VISITING;
128     for (LockNode n : children) {
129         if (n.hasCycle(visited, touchedNodes)) { }
130         return true;
131     }
132 }
133     visited[lockId] = VisitState.VISITED;
134 }
135 return false;
136 }
137
138 public Lock getLock() {
139     if (lock == null) lock = new ReentrantLock();
140     return lock;
141 }
142
143 public int getId() { return lockId; }
144 }

```

При виде сложного и длинного кода вы думаете, что не сможете написать его на собеседовании. Скорее всего, вас попросят написать схематичный псевдокод и, возможно, реализовать один из методов.

16.5. Предположим, что у вас есть 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: сначала — ThreadA, потом — ThreadB, потом — ThreadC. Разработайте механизм, гарантирующий, что сначала будет выполнен первый поток, после него — второй, а потом уже — третий.

Решение

Общий принцип заключается в проверке, завершился ли `first()` перед `second()` и завершился ли `second()` перед вызовом `third()`. Поскольку мы очень осторожны и заботимся о безопасности потоков, простых булевых флагов нам будет недостаточно.

А если использовать блокировку?

```

1 public class FooBad {
2     public int pauseTime = 1000;
3     public ReentrantLock lock1;
4     public ReentrantLock lock2;
5
6     public FooBad() {
7         try {
8             lock1 = new ReentrantLock();
9             lock2 = new ReentrantLock();
10            lock3 = new ReentrantLock();
11            // Получаем блокировку, чтобы избежать конфликтов
12            lock1.lock();
13            lock2.lock();
14            lock3.lock();
15        } catch (...) { ... } finally {
16    }
17
18    public void first() {
19        try {
20            ...
21            lock1.unlock(); // помечаем first() как завершенный
22        } catch (...) { ... }
23    }
24
25    public void second() {
26        try {
27            lock1.lock(); // ждем, пока завершится first()
28            lock1.unlock();
29            ...
30            lock2.unlock(); // помечаем second() как завершенный
31        } catch (...) { ... }
32    }
33
34
35    public void third() {
36        try {
37            lock2.lock(); // ждем, пока завершится second()
38            lock2.unlock();
39            ...
40        } catch (...) { ... }
41    }
42 }
```

Этот код не будет правильно работать из-за проблемы с *хозяином блокировки*. Один поток вызывает блокировку (конструктор `FooBad`), а другие потоки пытаются эту блокировку снять. Так поступать нельзя, в результате будет сгенерировано исключение. Право блокировки в Java принадлежит тому потоку, который ее вызвал.

Вместо этого мы можем использовать семафоры, логика работы останется той же:

```

1 public class Foo {
2     public Semaphore sem1;
3     public Semaphore sem2;
4     public Foo() {
5         try {
6             sem1 = new Semaphore(1);
7             sem2 = new Semaphore(1);
8             sem3 = new Semaphore(1);
9         } catch (...) { ... }
10    }
11    sem1.acquire();
12    sem2.acquire();
13    sem3.acquire();
14 } catch (...) { ... }
15 }
16
17 public void first() {
18     try {
19         ...
20         sem1.release();
21     } catch (...) { ... }
22 }
23
24 public void second() {
25     try {
26         sem1.acquire();
27         sem1.release();
28         ...
29         sem2.release();
30     } catch (...) { ... }
31 }
32
33 public void third() {
34     try {
35         sem2.acquire();
36         sem2.release();
37         ...
38     } catch (...) { ... }
39 }
40 }
```

- 16.6.** Дано: класс с синхронизированным методом A и обычным методом C. Если у вас есть два потока в одном экземпляре программы, могут ли они оба выполнить A одновременно? Могут ли они вызывать A и C одновременно?

Решение

Применяя слово synchronized к методу, мы гарантируем, что два потока не могут одновременно выполнить метод на том же самом объекте.

Это определяет наш ответ на первую часть вопроса. Если два потока являются одним и тем же экземпляром объекта, то они не могут одновременно выполнить метод A. Если же экземпляры объекта разные, то могут.

Вы можете увидеть это, изучая блокировки. Синхронизируемый метод блокирует метод в конкретном экземпляре объекта, запрещая любым потокам выполнять методы на том же самом объекте.

Два потока могут выполнять различные методы на объекте, так как блокировка относится к условию «метод + уровень объекта».

Хотя в задаче конкретно и не спрашивается об этом, обратите внимание, что если бы оба метода синхронизировались и были статическими, `thread1` не смог бы выполнить метод A, пока `thread2` выполнял метод B. Но это касается только статических (и синхронизированных) методов.

```

1 public void first() {
2     try {
3         lock1 = new ReentrantLock();
4         lock2 = new ReentrantLock();
5         lock3 = new ReentrantLock();
6         lock1.lock();
7         lock2.lock();
8         lock3.lock();
9         lock1.unlock(); // помечаем first() как завершенное
10    } catch (...) { ... }
11 }
12
13 public void second() {
14     try {
15         lock1.lock(); // если, пока завершался first()
16         lock1.unlock();
17         lock2.lock();
18         lock3.lock();
19         lock2.unlock(); // помечаем second() как завершенное
20     } catch (...) { ... }
21 }
22
23 public void third() {
24     try {
25         lock1.lock();
26         lock1.unlock();
27         lock2.lock();
28         lock3.lock();
29         lock2.unlock();
30     } catch (...) { ... }
31 }
32
33 public void fourth() {
34     try {
35         lock1.lock();
36         lock1.unlock();
37         lock2.lock();
38         lock3.lock();
39         lock2.unlock();
40     } catch (...) { ... }
41 }
42

```

по туном он блокирует, отрываясь от него, чтобы не блокировать этого же метода. Этот код не будет работать, так как методы не могут быть синхронизированы в один и тот же момент времени. Поэтому, если вы хотите использовать эту функцию, вам придется использовать блокировку в другом месте. Для этого надо просто поменять местами блокировки в `first()` и `second()`. Тогда блокировка `lock1` будет блокировать метод `first()`, а блокировка `lock2` — метод `second()`.

ДОПОЛНИТЕЛЬНЫЕ ЗАДАЧИ: РЕШЕНИЯ

Представим, что у нас есть сколько-нибудь большое количество чисел от 0 до 9 — пусть a_0 — это первое число (если $a_0 = 0$, то $a_0 = 1$), a_1 — второе и т. д. Пусть b_0 — исходное значение b , а b_1 — новое значение b . Давайте обозначим разницу $a_0 - b_0$ как $diff$.

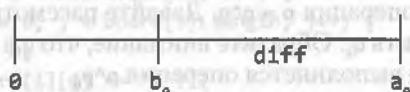
17. Задачи умеренной сложности

- 17.1.** Напишите функцию, меняющую местами значения переменных, не используя временные переменные.

Решение

Это классическая задача, которую любят предлагать на собеседованиях, и она достаточно проста. Пусть a_0 — это исходное значение a , а b_0 — исходное значение b . Обозначим $diff$ разницу $a_0 - b_0$.

Давайте покажем взаимное расположение всех этих значений на числовой оси для случая, когда $a > b$:



Присвоим a значение $diff$. Если сложить значение b и $diff$, то мы получим a_0 (результат следует сохранить в b). Теперь у нас $b = a_0$ и $a = diff$. Все, что нам остается сделать, — присвоить b значение $a_0 - diff$, а это значение представляет собой $b - a$.

Приведенный далее код реализует этот алгоритм:

```
1 public static void swap(int a, int b) {
2     // Пример для a = 9, b = 4
3     a = a - b; // a = 9 - 4 = 5
4     b = a + b; // b = 5 + 4 = 9
5     a = b - a; // a = 9 - 5
6
7     System.out.println("a: " + a);
8     System.out.println("b: " + b);
9 }
```

Можно решить эту задачу с помощью битовой манипуляции. Такой подход позволит нам работать с разными типами данных, а не только с `integer`.

```
1 public static void swap_opt(int a, int b) {
2     // Пример для a = 101 (в двоичной системе) и b = 110
3     a = a ^ b; // a = 101 ^ 110 = 011
4     b = a ^ b; // b = 011 ^ 110 = 101
5     a = a ^ b; // a = 011 ^ 101 = 110
6
7     System.out.println("a: " + a);
8     System.out.println("b: " + b);
9 }
```

Этот код использует операцию `xor`. Проще всего понять, как работает код, взглянув на два бита — p и q . Давайте обозначим как p_0 и q_0 исходные значения.

Если мы сможем поменять местами два бита, то алгоритм будет работать правильно. Давайте рассмотрим работу алгоритма пошагово:

```
1 p = p0^q0 /* 0 если p0 = q0, 1 если p0 != q0 */
2 q = p^q0 /* равно значению p0 */
3 p = p^q /* равно значению q0 */
```

В строке 1 выполняется операция $p = p_0 \oplus q_0$, результатом которой будет 0, если $p_0 = q_0$, и 1, если $p_0 \neq q_0$.

В строке 2 выполняется операция $q = p \oplus q_0$. Давайте проанализируем оба возможных значения p . Так как мы хотим поменять местами значения p и q , в результате должен получиться 0:

- $p = 0$: в этом случае $p_0 = q_0$, так как нам нужно вернуть p_0 или q_0 . XOR любого значения с 0 всегда дает исходное значение, поэтому результатом этой операции будет q_0 (или p_0).
- $p = 1$: в этом случае $p_0 \neq q_0$. Нам нужно получить 1, если $q_0 = 0$, и 0, если $p_0 = 1$. Именно такой результат получается при операции XOR любого значения с 1.

В строке 3 выполняется операция $p = p \oplus q$. Давайте рассмотрим оба значения p . В результате мы хотим получить q_0 . Обратите внимание, что q в настоящий момент равно p_0 , поэтому на самом деле выполняется операция $p \oplus p_0$.

- $p = 0$: так как $p_0 = q_0$, мы хотим вернуть p_0 или q_0 . Выполняя $0 \oplus p_0$, мы вернем p_0 (q_0).
- $p = 1$: выполняется операция $1 \oplus p_0$. В результате мы получаем инверсию p_0 , что нам и нужно, так как $p_0 \neq q_0$.

Остается только присвоить p значение q_0 , а q — значение p_0 . Мы удостоверились, что наш алгоритм корректно меняет местами каждый бит, а значит, результат будет правильным.

17.2. Разработайте алгоритм, проверяющий результат игры в крестики-нолики.

Решение

На первый взгляд эта задача кажется тривиальной — нам достаточно проверить доску. Разве это трудно? При ближайшем рассмотрении оказывается, что задача несколько сложнее и не имеет «идеального» решения. Выбор оптимального решения зависит от множества факторов.

Есть несколько принципиальных вопросов, которые следует решить:

- Будет `hasWon` вызываться однократно или несколько раз (например, как компонент веб-сайта с онлайн-версией игры)? Если планируется многократный вызов, нам, возможно, понадобится время на предварительную обработку, чтобы оптимизировать время выполнения `hasWon`.
- 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? Все хотят получить эффективный код, но важно, чтобы код был понятен и удобен в обслуживании.

Решение 1. Если hasWon вызывается много раз

Допустим, что у нас есть около 20 000 полей (3×3) для игры в крестики-нолики. Представим игровое поле как целое число, где каждый разряд соответствует клетке (0 — пусто, 1 — крестик (красный цвет), 2 — нолик (синий цвет)). Можно создать хэш-таблицу или массив со всеми возможными полями в качестве ключей и значений, указывающих, кто победил. Тогда функция будет простой:

```
1 public int hasWon(int board) {
2     return winnerHashtable[board];
3 }
```

Чтобы преобразовать поле (массив символов) в `int`, можно использовать систему счисления по основанию 3. Каждая доска представлена как $3^0 v_0 + 3^1 v_1 + 3^2 v_2 + \dots + 3^8 v_8$, где $v_i = 0$, если клетка пустая, 1 — нолик, 2 — крестик.

```
1 public static int convertBoardToInt(char[][] board) {
2     int factor = 1;
3     int sum = 0;
4     for (int i = 0; i < board.length; i++) {
5         for (int j = 0; j < board[i].length; j++) {
6             int v = 0;
7             if (board[i][j] == 'x') {
8                 v = 1;
9             } else if (board[i][j] == 'o') {
10                v = 2;
11            }
12            sum += v * factor;
13            factor *= 3;
14        }
15    }
16    return sum;
17 }
```

Теперь нахождение победителя превращается в задачу поиска в хэш-таблице.

Конечно, нам придется преобразовывать поле в этот формат каждый раз, когда мы хотим выяснить, кто является победителем. Это решение не позволит сэкономить время, по сравнению с другими возможными решениями. Но если мы будем хранить поле в таком формате с самого начала, процесс поиска станет эффективнее.

Решение 2. Поле размером 3×3

Если мы решаем задачу для поля 3×3 , код будет относительно коротким и простым. Самое сложное — сделать его наглядным и систематизировать, чтобы избежать дублирования фрагментов.

```
1 Piece hasWon1(Piece[][] board) {
2     for (int i = 0; i < board.length; i++) {
3         /* Проверяем ряды */
4         if (board[i][0] != Piece.Empty &&
5             board[i][0] == board[i][1] &&
6             board[i][0] == board[i][2]) {
7                 return board[i][0];
8             }
9     }
10 }
```

продолжение

```

9
10    /* Проверяем колонки */
11    if (board[0][i] != Piece.Empty &&
12        board[0][i] == board[1][i] &&
13        board[0][i] == board[2][i]) {
14            return board[0][i];
15        }
16    }
17
18    /* Проверяем диагональ */
19    if (board[0][0] != Piece.Empty &&
20        board[0][0] == board[1][1] &&
21        board[0][0] == board[2][2]) {
22            return board[0][0];
23        }
24
25    /* Проверяем обратную диагональ */
26    if (board[2][0] != Piece.Empty &&
27        board[2][0] == board[1][1] &&
28        board[2][0] == board[0][2]) {
29        return board[2][0];
30    }
31    return Piece.Empty;
32 }

```

Решение 3. Поле размером $N \times N$

Это решение можно считать расширением кода для поля 3×3 . (В дополнительных материалах вы найдете несколько других способов решения задачи.)

```

1 Piece hasWon3(Piece[][] board) {
2     int N = board.length;
3     int row = 0;
4     int col = 0;
5
6     /* Проверяем ряды */
7     for (row = 0; row < N; row++) {
8         if (board[row][0] != Piece.Empty) {
9             for (col = 1; col < N; col++) {
10                 if (board[row][col] != board[row][col-1]) break;
11             }
12             if (col == N) return board[row][0];
13         }
14     } // время на поиск
15
16     /* Проверяем колонки */
17     for (col = 0; col < N; col++) {
18         if (board[0][col] != Piece.Empty) {
19             for (row = 1; row < N; row++) {
20                 if (board[row][col] != board[row-1][col]) break;
21             }
22         }
23     }
24
25     /* Проверяем диагональ */
26     if (board[0][0] != Piece.Empty &&
27         board[0][0] == board[1][1] &&
28         board[0][0] == board[2][2]) {
29         return board[0][0];
30     }
31
32     /* Проверяем обратную диагональ */
33     if (board[2][0] != Piece.Empty &&
34         board[2][0] == board[1][1] &&
35         board[2][0] == board[0][2]) {
36         return board[2][0];
37     }
38
39     return Piece.Empty;
40 }

```

```

23     }
24 }
25
26 /* Проверяем диагональ (от верхнего левого угла до нижнего правого) */
27 if (board[0][0] != Piece.Empty) {
28     for (row = 1; row < N; row++) {
29         if (board[row][row] != board[row-1][row-1]) break;
30     }
31     if (row == N) return board[0][0];
32 }
33
34 /* Проверяем диагональ (от нижнего левого до верхнего правого) */
35 if (board[N-1][0] != Piece.Empty) {
36     for (row = 1; row < N; row++) {
37         if (board[N-row-1][row] != board[N-row][row-1]) break;
38     }
39     if (row == N) return board[N-1][0];
40 }
41
42 return Piece.Empty;
43 }

```

Независимо от того, как вы будете решать задачу, алгоритмы — это не самое сложное. Трудность состоит в том, чтобы создать чистый и удобный для дальнейшей работы код, что и будет оценивать интервьюер.

17.3. Напишите алгоритм, вычисляющий число конечных нулей в $n!$.

Решение

Самый простой подход — вычислить факториал и затем подсчитать, сколько конечных нулей присутствует в результате, повторяя операцию деления на 10. Но есть одна трудность — факториал быстро превысит возможности типа `int`. Чтобы избежать этой проблемы, нам нужно посмотреть на проблему с точки зрения математики.

Давайте проанализируем факториал:

$$19! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11 * 12 * 13 * 14 * 15 * 16 * 17 * 18 * 19$$

Конечный нуль появляется за счет умножения на 10, а умножение на 10 может получиться за счет перемножения чисел 5 и 2.

В $19!$ конечные нули создают следующие элементы:

$$19! = 2 * \dots * 5 * \dots * 10 * \dots * 15 * 16 * \dots$$

Чтобы подсчитать количество нулей, можно подсчитать количество пар произведений 5 и 2. Умножение на 2 будет встречаться существенно чаще, чем умножение на 5, поэтому достаточно ограничиться подсчетом умножений на 5.

Здесь есть один тонкий момент — число 15. Оно кратно 5 (а значит, добавляет один конечный 0). Число 25 добавит нам еще два нуля (потому что $25 = 5 * 5$).

Существует два способа написать код.

Первый способ — итерация от 2 до n и подсчет количества вхождений сомножителя 5 в каждое число.

```

1 /* Если в числе есть 5, возвращаем степень 5
2 * 5 -> 1,
3 * 25-> 2, и т. д.
4 */
5 public int factorsOf5(int n) {
6     int count = 0;
7     while (n % 5 == 0) {
8         count++;
9         n /= 5;
10    }
11    return count;
12 }
13
14 public int countFactZeros(int num) {
15     int count = 0;
16     for (int i = 2; i <= num; i++) {
17         count += factorsOf5(i);
18     }
19     return count;
20 }

```

Это неплохой код, но можно решить задачу более эффективно. Сначала мы подсчитаем, сколько чисел, кратных 5, находится между 1 и $n/5$, 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 public 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 }

```

Задача напоминает головоломку, но можно подойти к ее решению логически. Мы проанализировали задачу, выяснили, за счет чего появляются нули, и придумали решение. Вам следует придерживаться алгоритма, чтобы правильно решить задачу.

17.4. Напишите метод, находящий максимальное из двух чисел, не используя операторы `if-else` или любые другие операторы сравнения.

Решение

Самый распространенный вариант реализации функции `max` — проверка знака выражения $a - b$. В этом случае мы не можем использовать оператор сравнения, но можем использовать умножение.

Обозначим знак выражения $a - b$ как k . Если $a - b \geq 0$, то $k = 1$, иначе $k = 0$. Пусть c будет инвертированным значением k .

Код будет иметь вид:

```

1 /* Отражаем 1 в 0 и 0 в 1 */
2 public static int flip(int bit) {
3     return 1^bit;
4 }
5
6 /* Возвращаем 1, если число положительное, и 0, если отрицательное*/
7 public static int sign(int a) {
8     return flip((a >> 31) & 0x1);
9 }
10
11 public static int getMaxNaive(int a, int b) {
12     int k = sign(a - b);
13     int q = flip(k);
14     return a * k + b * q;
15 }
```

Это почти работоспособный код. Проблемы начинаются при переполнении. Предположим, что $a = \text{INT_MAX} - 2$ и $b = -15$. В этом случае $a - b$ перестанет помещаться в 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 понять, что a и b имеют разные знаки. Если у a и b разные знаки, то пусть $k = \text{sign}(a)$.

Логика будет следующей:

```

1 если у a и b разные знаки:
2 // если a > 0, то b < 0 и k = 1.
3 // если a < 0, то b > 0 и k = 0.
4 // так или иначе, k = sign(a)
5 пусть k = sign(a)
6 иначе
7 пусть k = sign(a - b) // переполнение невозможно
```

Приведенный далее код реализует этот алгоритм, используя умножение вместо операторов сравнения:

```

1 public static 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 a >= 1, то 1, иначе 0
6     int sc = sign(c); // зависит от переполнения a - b
7
8     /* Цель: найти k, которое = 1, если a > b, и 0, если a < b.
9      * если a = b, k не имеет значения */
10 }
```

продолжение ⤵

```

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); // отражение k
19
20    return a * k + b * q;
21 }

```

Отметим, что для большей наглядности мы разделяем код на методы и вводим переменные. Это не самый компактный или эффективный способ нависания кода, но так мы делаем код понятнее.

17.5. В игру «Гениальный отгадчик» играют следующим образом:

У компьютера есть четыре слота, в каждом слоте находится шар красного (R), желтого (Y), зеленого (G) или синего (B) цвета. Последовательность RGGB означает, что в слоте 1 — красный шар, в слотах 2 и 3 — зеленый, 4 — синий. Пользователь должен угадать цвета шаров, например предположить, что там YRGB. Если вы угадали правильный цвет, то вы получаете «хит». Если вы назвали цвет, который присутствует в раскладе, но находится в другом слоте, вы получаете «псевдохит». Слот с «хитом» не может быть назван «псевдохитом». Например, если фактический расклад RGYB, а ваше предположение GGRR, то ответ должен быть таким: один «хит» и один «псевдохит».

Напишите метод, который возвращает число «хитов» и «псевдохитов».

Решение

Задача довольно проста, но даже в ней легко допустить несколько ошибок. Вам нужно проверить свой код, учитывая все граничные случаи.

Давайте создадим массив, который будет хранить информацию о том, сколько раз каждый символ появлялся в solution (решении), исключая случаи, когда слот был «хитом». Затем мы просматриваем guess для подсчета количества псевдохитов.

Приведенный далее код реализует этот алгоритм:

```

1 public class Result {
2     public int hits = 0;
3     public int pseudoHits = 0;
4
5     public String toString() {
6         return "(" + hits + ", " + pseudoHits + ")";
7     }
8 }
9
10 public int code(char c) {
11     switch (c) {
12         case 'B':
13             return 0;

```

```

14     case 'G':
15         return 1;
16     case 'R':
17         return 2;
18     case 'Y':
19         return 3;
20     default:
21         return -1;
22     }
23 }
24
25 public static int MAX_COLORS = 4;
26
27 public Result estimate(String guess, String solution) {
28     if (guess.length() != solution.length()) return null;
29
30     Result res = new Result();
31     int[] frequencies = new int[MAX_COLORS];
32
33     /* Вычисляем хиты и строим таблицу частот */
34     for (int i = 0; i < guess.length(); i++) {
35         if (guess.charAt(i) == solution.charAt(i)) {
36             res.hits++;
37         } else {
38             /* Всего лишь увеличиваем таблицу частот (что
39             * будет использоваться для псевдохитов), если это не хит.
40             * Если это хит, слот уже будет "занят" */
41             int code = code(solution.charAt(i));
42             frequencies[code]++;
43         }
44     }
45
46     /* Вычисляем псевдохиты */
47     for (int i = 0; i < guess.length(); i++) {
48         int code = code(guess.charAt(i));
49         if (code >= 0 && frequencies[code] > 0) {
50             res.pseudoHits++;
51             frequencies[code]--;
52         }
53     }
54     return res;
55 }

```

Обратите внимание, что чем проще алгоритм, тем важнее написать чистый и правильный код. В этом случае мы выделили `code(char c)` в собственный метод, а также создали класс `Result` для хранения результата вместо обычного его вывода на консоль.

- 17.6.** Дано: массив целых чисел. Напишите метод, находящий индексы m и n такие, что для полной сортировки массива достаточно будет отсортировать элементы от m до n . Минимизируйте n и m (то есть найдите наименьшую такую последовательность).

Решение

Прежде чем приступить к решению, давайте убедимся, что мы понимаем, на что должен быть похож ответ. Если мы ищем пару индексов, это означает, что некоторая средняя часть массива будет отсортирована, а весь массив будет упорядочен.

Теперь давайте рассмотрим пример:

1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19

Первая мысль — найти самую длинную увеличивающуюся субпоследовательность в начале списка и самую длинную увеличивающуюся субпоследовательность в конце.

левая часть (left): 1, 2, 4, 7, 10, 11

средняя часть (middle): 7, 12

правая часть (right): 6, 7, 16, 18, 19

Эти субпоследовательности можно легко выявить. Мы начинаем слева и справа и движемся к центру. Когда встречается неупорядоченный элемент, это означает, что увеличивающаяся/уменьшающаяся субпоследовательности обнаружены.

Для решения задачи нам понадобится отсортировать среднюю часть массива, чтобы получить полностью упорядоченный массив. Таким образом, должны выполняться следующие условия:

```
/* все элементы слева меньше, чем все элементы в середине */
min(middle) > end(left)
/* все элементы средней части меньше, чем все элементы в правой */
max(middle) < start(right)
```

Или, другими словами:

`left < middle < right`

Фактически, это условие никогда не будет соблюдаться. Середина по определению неупорядочена: `left.end > middle.start` и `middle.end > right.start`. То есть вам не достаточно просто отсортировать середину, чтобы упорядочить весь массив.

Но мы можем уменьшать левые и правые субпоследовательности, пока исходные условия не станут соблюдаться.

Пусть `min = min(middle)` и `max = max(middle)`.

Слева мы начинаем двигаться от конца субпоследовательности (значение 11, элемент 5) и смещаемся на один элемент влево. Как только обнаруживается элемент `i`, такой что `array[i] < min`, это означает, что можно сортировать середину и что та часть массива упорядочена.

Затем мы проделываем аналогичные действия справа. Максимальное значение — 12. Мы начинаем с правой субпоследовательности (значение 6) и перемещаемся на шаг правее. Мы сравниваем `max` (12) с 6, затем с 7, затем с 16. Когда мы достигли 16, то знаем, что не существует элементов, меньших чем 12 (увеличивающаяся субпоследовательность). Теперь можно отсортировать середину, чтобы сделать весь массив упорядоченным.

Следующий код реализует этот алгоритм:

```
1 int findEndOfLeftSubsequence(int[] array) {
2     for (int i = 1; i < array.length; i++) {
3         if (array[i] < array[i - 1]) return i - 1;
4     }
}
```

```

5     return array.length - 1;
6 }
7
8 int findStartOfRightSubsequence(int[] array) {
9     for (int i = array.length - 2; i >= 0; i--) {
10         if (array[i] > array[i + 1]) return i + 1;
11     }
12     return 0;
13 } Функция
14
15 int shrinkLeft(int[] array, int min_index, int start) { от этого момента и до конца функции
16     int comp = array[min_index];
17     for (int i = start - 2; i >= 0; i--) {
18         if (array[i] <= comp) return i + 1;
19     }
20     return 0;
21 } состоит из 18 строк
22
23 int shrinkRight(int[] array, int max_index, int start) { и 18 строк
24     int comp = array[max_index];
25     for (int i = start; i < array.length; i++) {
26         if (array[i] >= comp) return i - 1;
27     }
28     return array.length - 1;
29 } и 18 строк
30
31 void findUnsortedSequence(int[] array) { все строки
32     /* находим левую субпоследовательность */
33     int end_left = findEndOfLeftSubsequence(array);
34
35     /* находим правую субпоследовательность */
36     int start_right = findStartOfRightSubsequence(array);
37
38     /* находим минимальный и максимальный элементы середины */
39     int min_index = end_left + 1;
40     if (min_index >= array.length) return; // Уже отсортирован
41
42     int max_index = start_right - 1;
43     for (int i = end_left; i <= start_right; i++) {
44         if (array[i] < array[min_index]) min_index = i;
45         if (array[i] > array[max_index]) max_index = i;
46     } все строки
47
48     /* понижаем левую, пока меньше, чем [min_index] */
49     int left_index = shrinkLeft(array, min_index, end_left); все строки
50
51     /* понижаем правую, пока больше, чем array[max_index] */
52     int right_index = shrinkRight(array, max_index, start_right); все строки
53
54     System.out.println(left_index + " " + right_index); все строки
55 }

```

Обратите внимание, что мы использовали множество методов. Можно было бы ограничиться одним большим методом, но это сделает код более трудным для восприятия, поддержки и проверки. На собеседовании нужно учитывать все эти аспекты.

17.7. Дано: целое число. Выведите его значение прописью (например, «одна тысяча двести тридцать четыре»).

Решение

Это несложная, но несколько утомительная задача. Вам нужно сконцентрироваться на решении и хорошо его протестировать.

Можно преобразовать число вроде 19 323 984 в несколько трехразрядных сегментов с соответствующими добавками «thousand» и «million». Например¹:

```
convert(19,323,984) = convert(19) + " million " +
+ convert(323) + " thousand " +
+ convert(984)
```

Следующий код реализует этот алгоритм:

```
1 public String[] digits = {"One", "Two", "Three", "Four", "Five",
2     "Six", "Seven", "Eight", "Nine"};
3 public String[] teens = {"Eleven", "Twelve", "Thirteen",
4     "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen",
5     "Nineteen"};
6 public static String[] tens = {"Ten", "Twenty", "Thirty", "Forty",
7     "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
8 public static String[] bigs = {"", "Thousand", "Million"};
9
10    public static String numToString(int number) {
11        if (number == 0) {
12            return "Zero";
13        } else if (number < 0) {
14            return "Negative " + numToString(-1 * number);
15        }
16
17        Int count = 0;
18        String str = "";
19
20        while (number > 0) {
21            if (number % 1000 != 0) {
22                str = numToString100(number % 1000) + bigs[count] + " " + str;
23            }
24            number /= 1000;
25            count++;
26        }
27    }
28
29    return str;
```

¹ Листинг приводится без перевода — как было в оригинале, чтобы не переписывать весь код. Для создания аналогичной русскоязычной программы код придется несколько изменить. — Примеч. перев.

```

30 }
31
32 public static String numToString100(int number) {
33     String str = "";
34
35     /* Конвертируем сотни */
36     if (number >= 100) {
37         str += digits[number / 100 - 1] + " Hundred ";
38         number %= 100;
39     }
40
41     /* Конвертируем десятки */
42     if (number >= 11 && number <= 19) {
43         return str + teens[number - 11] + " ";
44     } else if (number == 10 || number >= 20) {
45         str += tens[number / 10 - 1] + " ";
46         number %= 10;
47     }
48
49     /* Конвертируем единицы */
50     if (number >= 1 && number <= 9) {
51         str += digits[number - 1] + " ";
52     }
53
54     return str;
55 }

```

Важно убедиться, что вы проверили все граничные случаи, а в этой задаче их много.

17.8. Дано: массив целых чисел (положительных и отрицательных). Найдите непрерывную последовательность с самой большой суммой. Возвратите сумму.

Решение

Это довольно сложная, но очень популярная задача. Давайте решим ее на примере массива:

2 3 -8 -1 2 4 -2 3

Если рассматривать массив как содержащий чередующиеся последовательности положительных и отрицательных чисел, то не имеет смысла рассматривать части положительных или отрицательных субпоследовательностей. Почему? Включая часть отрицательной субпоследовательности, мы уменьшаем итоговое значение суммы, значит, нам не стоит включать часть отрицательной субпоследовательности вообще. Включение части положительной субпоследовательности выглядит еще более странным, поскольку включение этой субпоследовательности целиком всегда даст больший результат.

Нужно придумать алгоритм, рассматривая массив как последовательность отрицательных и положительных чисел, расположенных вперемежку. Любое число можно представить в виде суммы субпоследовательностей положительных и отрицательных чисел. В нашем примере массив можно сократить до:

5 -9 6 -2 3

Мы еще не получили отличный алгоритм, но теперь лучше понимаем, с чем имеем дело.

Рассмотрим предыдущий массив. Нужно ли учитывать субпоследовательность $\{5, -9\}$? В сумме мы получим -4 , 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 субпоследовательности, сумма каждой из которых больше, чем вклад отрицательной величины.

Давайте продвигаться начиная с первого элемента в массиве.

5 — это самая большая сумма, встретившаяся нам. Таким образом, `maxsum=5` и `sum=5`. Затем мы видим следующее число (-9) . Если добавить это число к `sum`, то получится отрицательная величина. Нет смысла расширять субпоследовательность с 5 до -9 (-9 уменьшает общую сумму до -4). Таким образом, мы просто сбрасываем значение `sum`.

Теперь мы дошли до следующего элемента (6) . Эта субпоследовательность больше, чем 5 , таким образом, мы обновляем значения `maxsum` и `sum`.

Затем мы смотрим на следующий элемент (-2) . Добавление этого числа к 6 сделает `sum=4`. Так как это не окончательное значение, наша субпоследовательность выглядит как $\{6, -2\}$. Мы обновляем `sum`, но не `maxsum`.

Наконец мы смотрим на следующий элемент (3) . Добавление 3 к `sum` (4) даст нам 7 , таким образом, мы обновляем `maxsum`. Максимальная последовательность имеет вид $\{6, -2, 3\}$.

Когда мы работаем с развернутым массивом, логика остается такой же. Следующий код реализует этот алгоритм:

```

1 public static 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 отрицательных чисел? Как действовать в этом случае? Рассмотрим 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` (для случая ошибки).

В нашем коде был использован второй ответ (`sum=0`), но в этом вопросе не существует однозначного «правильного» решения. Обсудите это с интервьюером.

- 17.9.** Разработайте метод, вычисляющий частоту использования заданного слова в книге.

Решение

Прежде всего, вы должны уточнить, сколько раз будет выполняться эта операция — только один раз или многократно. Понадобится найти частоту употребления слова *dog* и завершить работу или нужно будет проанализировать слово *dog*, а затем *cat*, *mouse* и т. д.

Решение. Однократный запрос

В этом случае мы просто просматриваем книгу пословно и считаем, сколько раз употребляется конкретное слово. Это займет $O(n)$ времени. Лучших результатов добиться нельзя, так как необходимо проверить каждое слово в книге.

Решение. Повторяющиеся запросы

Если мы собираемся повторять поиски слов, следует выделить дополнительное время и память, чтобы провести предварительную обработку данных. Например, создать хэш-таблицу, которая будет ставить соответствие между словами и частотой их употребления. Тогда любое слово можно будет проанализировать за $O(1)$ времени. Код этого решения приведен ниже:

```

1 Hashtable<String, Integer> setupDictionary(String[] book) {
2     Hashtable<String, Integer> table =
3         new Hashtable<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(Hashtable<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 }
```

Обратите внимание, что, поскольку задача достаточно простая, интервьюер обратит внимание, насколько аккуратно вы ее решите.

- 17.10. Поскольку XML излишне «многословен», закодируйте его так, чтобы каждый тег преобразовывался в какое-либо целое значение:

```

Element --> Tag Attributes END Children END
Attribute --> Tag Value
END --> 0
Tag --> любое предопределенное целое значение
Value --> string value END

```

Преобразуем XML-код в сжатую форму, подразумевая, 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 как Element и Attribute, наш код будет предельно прост. При реализации решения мы используем древовидную структуру. В процессе работы будет неоднократно вызываться encode(), обрабатывающий код немного различающимися способами в зависимости от типа элемента XML.

```

1 public static void encode(Element root, StringBuffer 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 public static void encode(String v, StringBuffer sb) {
18     sb.append(v);
19     sb.append(" ");
20 }
21
22 public static void encode(Attribute attr, StringBuffer sb) {
23     encode(attr.getTagCode(), sb);
24     encode(attr.value, sb);
25 }
26

```

```

27 public static String encodeToString(Element root) {
28     StringBuffer sb = new StringBuffer();
29     encode(root, sb);
30     return sb.toString();
31 }

```

Обратите внимание, в строке 17 используется метод `encode` для строки. В этом нет необходимости, все, что он делает, — вставляет строку и пробел после нее. Но это позволяет нам гарантировать, что после каждого элемента будет вставлен пробел. Если этого не сделать, можно легко повредить декодированию, просто забыв вставить пустую строку.

- 17.11.** Реализуйте метод `rand7()` на базе метода `rand5()`. Другими словами, имеется метод, генерирующий случайные числа в диапазоне от 0 до 4 (включительно). Напишите использующий его метод, генерирующий случайное число в диапазоне от 0 до 6 (включительно).

Решение

Для правильной реализации этой функции следует помнить, что вероятность появления любого числа из диапазона от 0 до 6 составляет $1/7$.

Первая попытка (фиксированное число вызовов)

Первый шаг — попробовать сгенерировать все числа в диапазоне от 0 до 8 и затем вычислить остаток от деления на 7. Наш код будет иметь вид:

```

1 int rand7() {
2     int v = rand5() + rand5();
3     return v % 7;
4 }

```

К сожалению, этот код не обеспечит равную вероятность значений. Это становится заметно, если взглянуть на результаты каждого вызова `rand5()`, сравнив их с результатами функции `rand7()`.

1-й вызов	2-й вызов	Результат
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
1	0	1
1	1	2
1	2	3
1	3	4
1	4	5
2	0	2
2	1	3
2	2	4

1-й вызов	2-й вызов	Результат
2	3	5
2	4	6
3	0	3
3	1	4
3	2	5
3	3	6
3	4	0
4	0	4
4	1	5
4	2	6
4	3	0
4	4	1

Вероятность появления каждой строки — $1/25$, так как `rand5()` вызывается дважды (вероятность генерации каждого числа составляет $1/5$). Если вы подсчитаете частоту появления каждого числа, то обратите внимание, что функция `rand()` будет возвращать 0 с вероятностью $5/25$, но 0 должен генерироваться с вероятностью $3/5$. Это означает, что наша функция работает неправильно, и на самом деле мы не можем добиться вероятности $1/7$.

Допустим, что мы изменим функцию так, чтобы использовать оператор `if`, изменить постоянный коэффициент или добавить еще один вызов `rand5()`. Мы опять получим такую же таблицу, но вероятность каждой из строк будет $1/5^k$, 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:

$$P(\text{rand7}() = 6) = 1/5^1 + 1/5^2 + \dots + 1/5^n$$

Мы знаем, что в нашем случае эта вероятность должна составлять $1/7$.

Но это невозможно, поскольку 5 и 7 — взаимно простые числа, 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()`.

Но задачу все еще можно решить. Нам придется использовать цикл `while` и смириться с тем, что невозможно указать точное количество итераций для получения результата.

Вторая попытка (неопределенное количество вызовов)

Как только нам разрешили использовать цикл `while`, работа становится намного проще. Сгенерируем множество равновероятных значений. Если мы сможем сделать это, то будет достаточно отбросить элементы, превышающие наибольший, кратный семи, а оставшиеся поделить с остатком на 7 (операция `mod`). Это даст нам равновероятные значения в диапазоне от 0 до 6.

В приведенном далее коде мы используем диапазон от 0 до 24, выполняем $5 * \text{rand5}() + \text{rand5}()$. Затем отбрасываем значения из интервала 21–24, так как они нам не нужны, а затем делим каждое значение на 7 (с остатком), чтобы получить равновероятные значения от 0 до 6.

Обратите внимание, что поскольку мы отбрасываем значения, нет никакой гарантии, что `rand5()` возвратит нужное значение. Поэтому у нас используется неопределенное количество вызовов:

```

1 public static int rand7() {
2     while (true) {
3         int num = 5 * rand5() + rand5();
4         if (num < 21) {
5             return num % 7;
6         }
7     }
8 }
```

Обратите внимание, что $5 * \text{rand5}() + \text{rand5}()$ позволяет нам получить равномерное распределение чисел в диапазоне от 0 до 24. Это гарантирует, что каждое значение одинаково вероятно.

Можно ли вместо этого использовать $2 * \text{rand5}() + \text{rand5}()$? Нет, потому что полученные значения не будут равномерно распределены: можно получить 6 двумя разными способами ($6 = 2*1+4$ и $6 = 2*2+2$), а ноль — только одним ($0 = 2*0+0$). Таким образом, нарушится равномерное распределение значений в диапазоне.

Существует способ, который позволяет использовать $2 * \text{rand5}()$ и получить равномерное распределение, но он значительно сложнее:

```

1 public int rand7() {
2     while (true) {
3         int r1 = 2 * rand5();      /* четные между 0 и 9 */
4         int r2 = rand5();        /* используется для генерации 0 или 1 */
5         if (r2 != 4) {           /* в r2 есть дополнительные четные */
6             int rand1 = r2 % 2;  /* генерируем 0 или 1 */
7             int num = r1 + rand1; /* будет в диапазоне от 0 до 9 */
8             if (num < 7) {
9                 return num;
10            }
11        }
12    }
13 }
```

Существует бесконечное множество диапазонов, которые можно использовать. Удостоверьтесь, что диапазон достаточно велик и все значения равновероятны.

17.12. Разработайте алгоритм, обнаруживающий в массиве все пары целых чисел, сумма которых равна заданному значению.

Решение

Эту задачу можно решить двумя способами. Выбор определяется компромиссом между эффективностью использования времени, памяти или сложностью кода.

Простое решение

Очень простое и эффективное (по времени) решение — создание хэш-таблицы, отображающей целое число в целое число. Данный алгоритм работает, пошагово проходя весь массив. Для каждого элемента x в хэш-таблице ищется $\text{sum} - x$ и, если запись существует, выводится $(x, \text{sum} - x)$. После этого x добавляется в хэш-таблицу и проверяется следующий элемент.

Альтернативное решение

Давайте начнем с формулировки. Если мы попытаемся найти пару чисел, сумма которых равна z , то дополнение x будет $z - x$ (величина, которую нужно добавить к x , чтобы получить z). Если мы попытаемся найти пару чисел, при суммировании которых получается 12, дополнением к -5 будет число 17.

Представьте, что у нас есть 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 к first , мыдвигаем last назад, пока не найдем искомую величину. Если $\text{first} + \text{last} < \text{sum}$, то

дополнения к `first` не существует. Можно также перемещать `first` на встречу к `last`. Тогда мы остановимся, если `first` окажется больше, чем `last`.

Почему такое решение найдет все дополнения к `first`? Поскольку массив отсортирован, мы проверяем меньшие числа. Когда `first + last` меньше `sum`, нет смысла проверять меньшие значения, они не помогут найти дополнение.

Почему данное решение найдет все дополнения `last`? Потому что все пары формируются с помощью `first` и `last`. Мы нашли все дополнения `first`, а значит, нашли все дополнения `last`.

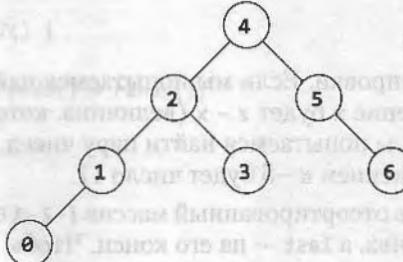
```

1 void printPairSums(int[] array, int sum) {
2     Arrays.sort(array);
3     int first = 0;
4     int last = array.length - 1;
5     while (first < last) {
6         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 }
```

- 17.13.** Дано: простая графообразная структура данных `BiNode`, содержащая указатели на два других узла. 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 работа алгоритма должна осуществляться в исходной структуре данных.

Решение

Эту на первый взгляд сложную задачу можно изящно решить с помощью рекурсии. Вы должны очень хорошо понимать механизмы рекурсии, чтобы ее использовать. Давайте рассмотрим простое бинарное дерево поиска:



Метод `convert` преобразует его в двусвязный список:

`0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6`

Давайте рекурсивно пройдемся по нему 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 результирующий связный список?

Да! Мы просто объединим части.

Псевдокод будет иметь вид:

```
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; // front of left
6 }
```

Чтобы реализовать основные элементы этого решения, нам понадобятся голова и хвост каждого связанного списка. Это можно сделать несколькими способами.

Решение 1. Дополнительная структура данных

Первый, более легкий подход — создание новой структуры данных `NodePair`, которая будет хранить только голову и хвост связного списка. Метод `convert` сможет возвратить результат типа `NodePair`.

Приведенный далее код реализует данный подход:

```
1 private class NodePair {
2     BiNode head;
3     BiNode tail;
4
5     public NodePair(BiNode head, BiNode tail) {
6         this.head = head;
7         this.tail = tail;
8     }
9 }
10
11 public NodePair convert(BiNode root) {
12     if (root == null) {
13         return null;
14     }
15
16     NodePair part1 = convert(root.node1);
17     NodePair part2 = convert(root.node2);
18
19     if (part1 != null) {
20         concat(part1.tail, root);
21     }
22
23     if (part2 != null) {
24         concat(root, part2.head);
```

продолжение ↗

```

25     }
26
27     return new NodePair(part1 == null ? root : part1.head,
28                           part2 == null ? root : part2.tail);
29 }
30
31 public static void concat(BiNode x, BiNode y) {
32     x.node2 = y;
33     y.node1 = x;
34 }

```

Этот код по-прежнему преобразовывает структуру данных `BiNode` на месте. Мы используем `NodePair` только как способ возврата дополнительных данных. С другой стороны, для этих же целей можно использовать двухэлементный массив `BiNode`, но это будет более «грязное» решение (а интервьюерам нравится чистый код).

Тем не менее нужно постараться минимально использовать дополнительные структуры данных.

Решение 2. Получение хвоста

Вместо возврата головы и хвоста связного списка с `NodePair` можно возвращать только голову, а затем использовать ее для обнаружения хвоста:

```

1  public static BiNode convert(BiNode root) {
2      if (root == null) {
3          return null;
4      }
5
6      BiNode part1 = convert(root.node1);
7      BiNode part2 = convert(root.node2);
8
9      if (part1 != null) {
10         concat(getTail(part1), root);
11     }
12
13     if (part2 != null) {
14         concat(root, part2);
15     }
16
17     return part1 == null ? root : part1;
18 }
19
20 public static BiNode getTail(BiNode node) {
21     if (node == null) return null;
22     while (node.node2 != null) {
23         node = node.node2;
24     }
25     return node;
26 }

```

Кроме `getTail`, этот код практически совпадает с предыдущим решением. Это не очень эффективно. `getTail` будет d раз обрабатывать вершину 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

Мы можем придумать еще один способ, отличающийся от предыдущего.

Нам понадобятся голова и хвост связного списка, получаемые с помощью BiNode. Но можно возвращать каждый список как голову кольцевого связного списка. Чтобы получить хвост, достаточно просто обратиться к `head.node1`.

```

1 public static BiNode convertToCircular(BiNode root) {
2     if (root == null) {
3         return null;
4     }
5
6     BiNode part1 = convertToCircular(root.node1);
7     BiNode part3 = convertToCircular(root.node2);
8
9     if (part1 == null && part3 == null) {
10        root.node1 = root;
11        root.node2 = root;
12        return root;
13    }
14    BiNode tail3 = (part3 == null) ? null : part3.node1;
15
16    /* подсоединение левой части к корню */
17    if (part1 == null) {
18        concat(part3.node1, root);
19    } else {
20        concat(part1.node1, root);
21    }
22
23    /* подсоединение правой части к корню */
24    if (part3 == null) {
25        concat(root, part1);
26    } else {
27        concat(root, part3);
28    }
29
30    /* соединение левой и правой частей */
31    if (part1 != null && part3 != null) {
32        concat(tail3, part1);
33    }
34
35    return part1 == null ? root : part1;
36 }
37
38 /* Конвертируем список в кольцевой связный список, а затем разрываем
39 * кольцевое соединение. */
40 public static BiNode convert(BiNode root) {
41     BiNode head = convertToCircular(root);

```

продолжение ↗

```

42     head.node1.node2 = null;
43     head.node1 = null;
44 }
45 }

```

Обратите внимание, что мы переместили основные части кода в `convertToCircular`. `convert` вызывает этот метод для получения головы кольцевого связного списка, а затем разрывает кольцевую связь.

Данный подход занимает $O(N)$ времени, так как каждый узел обрабатывается в среднем один раз (или, более точно, $O(1)$ раз).

- 17.14.** *O, нет! Вы только что закончили длинный документ, но сделали неудачную операцию Поиск/Замена. Вы случайно удалили все пробелы, пунктуацию и все прописные буквы. Предложение I reset the computer. It still didn't boot! превратилось в iresetthecomputeritstilldidntboot. После того как вы разделите строку на отдельные слова, можно будет восстановить пунктуацию слова. Большинство слов, кроме имен собственных, находятся в словаре.*

Для заданного словаря (списка слов) разработайте алгоритм, выполняющий оптимальную деконкатенацию строки. Алгоритм должен минимизировать число нераспознанных последовательностей символов.

Строчку "jesslookedjustliketimherbrother" необходимо оптимальным образом преобразовать в "JESS looked just like TIM her brother". Парсер не справился с семью символами, которые были написаны в верхнем регистре.

Решение

У интервьюеров есть любимые задачи. Иногда они предоставляют слишком много ненужной информации, как в нашем случае. Поэтому давайте разберемся, о чём идет речь.

Задача на самом деле посвящена способу разбиения строки на слова так, чтобы за рамками работы парсера оказывалось минимум слов.

Обратите внимание, что мы не пытаемся «понять» строку. Мы обрабатываем строку `thisisawesome` и получаем `this is a we some` или `this is awesome`.

Ключ к этой задаче — проанализировать строку с точки зрения подзадач. В каждой точке под оптимальным парсингом мы будем понимать лучшее из двух возможных решений:

1. Вставка пробела после этого символа.
2. Запрет вставки пробела после этого символа.

Давайте попробуем решить эту задачу на примере строки `thit`. При этом мы будем использовать следующие соглашения:

- ПРОПИСНЫЕ буквы обозначают недопустимые (отсутствующие в словаре) слова.
- Подчеркнутые — допустимые слова (есть в словаре).
- Полужирные — смежные символы (между ними нет пробелов).

Полужирные символы учитываются «в процессе обработки», мы еще не приняли решения, есть они в словаре или нет.

```

1 p(hit)
2   = min(T + p(hit), p(hit)) --> 1 inv.
3     T + p(hit) = min(T + H + p(it), T + p(hit)) --> 1 inv.
4       T + H + p(it) = min(T + H + i + p(t), T + H + p(it)) --> 2
5         T + H + i + p(t) = T + H + i + T = 3 invalid
6         T + H + p(it) = T + H + it = 2 invalid
7       T + p(hit) = min(T + hi + p(t), T + p(hit)) --> 1 inv.
8         T + hi + p(t) = T + hi + T = 2 invalid
9         T + p(hit) = T + hit = 1 invalid
10        p(hit) = min(TH + p(it), p(hit)) --> 2 inv.
11          TH + p(it) = min(TH + i + p(t), TH + p(it)) --> 2 inv.
12            TH + i + p(t) = TH + i + T = 3 invalid
13            TH + p(it) = TH + it = 2 invalid
14          p(hit) = min(THI + p(t), p(hit)) --> 4 inv.
15            THI + p(t) = THI + T = 4 invalid
16            p(hit) = THIT = 4 invalid

```

Обратите внимание, что каждый уровень делится на два этапа — сначала строка разбивается, а потом соединяется.

Например, когда мы вызываем `p(this)`, текущий обрабатываемый символ — первая буква — `t`. Мы рекурсивно движемся в двух различных направлениях. Сначала (строка 3) вставляем пробел после `t` и пытаемся найти оптимальный способ парсинга `hit`. Затем (строка 10) пытаемся найти лучший способ парсинга, если между `t` и `h` нет пробелов. Поскольку мы повторяем эту операцию многократно, проанализируем все возможные способы парсинга строки.

Приведенный далее код реализует это решение, алгоритм будет возвращать только количество неправильных символов:

```

1 public int parseSimple(int wordStart, int wordEnd) {
2     if (wordEnd >= sentence.length()) {
3         return wordEnd - wordStart;
4     }
5
6     String word = sentence.substring(wordStart, wordEnd + 1);
7
8     /* разбиваем текущее слово */
9     int bestExact = parseSimple(wordEnd + 1, wordEnd + 1);
10    if (!dictionary.contains(word)) {
11        bestExact += word.length();
12    }
13
14    /* расширяем текущее слово */
15    int bestExtend = parseSimple(wordStart, wordEnd + 1);
16
17    /* находим лучшее */
18    return Math.min(bestExact, bestExtend);
19 }

```

У данного алгоритма есть резервы оптимизации:

- Некоторые рекурсивные случаи перекрываются. Мы должны кэшировать только однократно. Как только элемент получен, его можно использовать повторно. Для этого нам понадобится динамическое программирование.

- В некоторых случаях можно предсказать, что результат конкретного парсинга возвращает некорректные строки. Например, если мы будем анализировать строку `xten`, то заметим, что не существует английских слов, которые начинаются с `xt`. Данный алгоритм, однако, попытается разбить строку как `xt + en` (`en`), `xte + n` (`n`) и `xten`. Каждый раз будет обнаружено, что таких слов в словаре нет. Вместо этого можно добавить условие — поместить пробел после `x` и улучшить парсинг.

Следующий код реализует обе оптимизации:

```

1 public int parseOptimized(int wordStart, int wordEnd, HT<Integer, Integer> cache) {
2     if (wordEnd >= sentence.length()) {
3         return wordEnd - wordStart;
4     }
5     if (cache.containsKey(wordStart)) {
6         return cache.get(wordStart);
7     }
8
9     String currentWord = sentence.substring(wordStart, wordEnd + 1);
10
11    /* проверяем префикс в словаре (false --> частичное совпадение) */
12    boolean validPartial = dictionary.contains(currentWord, false);
13
14    /* разбиваем текущее слово */
15    int bestExact = parseOptimized(wordEnd + 1, wordEnd + 1, cache);
16
17    /* если полная строка не в словаре, */
18    /* добавить к недопустимому счетчику */
19    if (!validPartial || !dictionary.contains(currentWord, true)) {
20        bestExact += currentWord.length();
21    }
22
23    /* расширяем текущее слово */
24    int bestExtend = Integer.MAX_VALUE;
25    if (validPartial) {
26        bestExtend = parseOptimized(wordStart, wordEnd + 1, cache);
27    }
28
29    /* находим лучшее */
30    int min = Math.min(bestExact, bestExtend);
31    cache.put(wordStart, min); // результат кэширования
32    return min;
33 }
```

Обратите внимание, что мы используем хэш-таблицу, чтобы кэшировать результаты. Ключ — начало слова. Таким образом, кэширование — лучший способ проанализировать остальную часть строки.

Можно написать код, возвращающий всю проанализированную строку, но это существенно более сложная задача. Мы используем интерфейсный класс `Result`, что-

бы возвращать и количество недопустимых символов, и оптимальную строку. Если реализовывать его на C++, то можно просто передать значение ссылкой:

```
1 public class Result {  
2     public int invalid = Integer.MAX_VALUE;  
3     public String parsed = "";  
4     public Result(int inv, String p) {  
5         invalid = inv;  
6         parsed = p;  
7     }  
8  
9     public Result clone() {  
10        return new Result(this.invalid, this.parsed);  
11    }  
12  
13    public static Result min(Result r1, Result r2) {  
14        if (r1 == null) {  
15            return r2;  
16        } else if (r2 == null) {  
17            return r1;  
18        }  
19        return r2.invalid < r1.invalid ? r2 : r1;  
20    }  
21 }  
22  
23 public Result parse(int wordStart, int wordEnd,  
24 Hashtable<Integer, Result> cache) {  
25     if (wordEnd >= sentence.length()) {  
26         return new Result(wordEnd - wordStart,  
27             sentence.substring(wordStart).toUpperCase());  
28     }  
29     if (cache.containsKey(wordStart)) {  
30         return cache.get(wordStart).clone();  
31     }  
32     String currentWord = sentence.substring(wordStart, wordEnd + 1);  
33     boolean validPartial = dictionary.contains(currentWord, false);  
34     boolean validExact = validPartial &&  
35     dictionary.contains(currentWord, true);  
36  
37     /* разбиваем текущее слово */  
38     Result bestExact = parse(wordEnd + 1, wordEnd + 1, cache);  
39     if (validExact) {  
40         bestExact.parsed = currentWord + " " + bestExact.parsed;  
41     } else {  
42         bestExact.invalid += currentWord.length();  
43         bestExact.parsed = currentWord.toUpperCase() + " " +  
44         bestExact.parsed;  
45     }  
46 }
```

продолжение ↴

```

47  /* расширяем текущее слово */
48  Result bestExtend = null;
49  if (validPartial) {
50      bestExtend = parse(wordStart, wordEnd + 1, cache);
51  }
52
53  /* находим лучшее */
54  Result best = Result.min(bestExact, bestExtend);
55  cache.put(wordStart, best.clone());
56  return best;
57 }

```

Осторожно используйте кэширование объектов в задачах с динамическим программированием. Если кэшируемое значение является объектом, а не примитивным типом данных, вероятно, вам придется клонировать объект. Обратите внимание на строки 30 и 55. Если мы не клонируем объект, то будущие вызовы `parse` будут изменять значения в кэше.

```

9
10 string currentWord = sentence.substring(wordStart, wordEnd + 1);
11
12 /* проверим правильность слова */
13 boolean validPartial = dictionary.contains(currentWord, false);
14
15 /* расширяем текущее слово */
16 int bestExact = validPartial ? currentWord.length() : 0;
17
18 /* если полная строка не в словаре, */
19 /* добавить к недопустимому фрагменту - вадимон */
20 /* добавить к недопустимому фрагменту - вадимон */
21 if (!validPartial || !dictionary.contains(currentWord, true)) {
22     bestExact += currentWord.length();
23 }
24
25 /* расширим текущее слово */
26 int bestExtend = dictionary.contains(currentWord + "вадимон", true)
27
28 /* если фрагмент недопустим, то */
29 /* добавить к недопустимому фрагменту - вадимон */
30 /* добавить к недопустимому фрагменту - вадимон */
31 /* добавить к недопустимому фрагменту - вадимон */
32 bestExtend += dictionary.contains(currentWord + "вадимон", true);
33
34
35 /* находим лучшее */
36 int min = Math.min(bestExact, bestExtend);
37 cache.put(wordStart, min); // результат кэширования
38 return min; // wordStart + 1 в диапазоне = bestExt
39 }
40 }

```

Обратите внимание на то что при кэшировании мы используем `clone()`. Ключ — начало слова. Таким образом, кэширование не будет загружать оставшуюся часть строки.

Можно записать код, воспринимающий всю предоставленную строку, не используя для хранения кэша. Мы используем интерфейсный класс `Result`.

18. Задачи повышенной сложности

- 18.1.** Напишите функцию суммирования двух чисел без использования `<+>` и других арифметических операторов.

Решение

Первое, что приходит в голову, — обработка битов. Почему? У нас нет выбора — нельзя использовать оператор `<+>`. Так что будем суммировать числа так, как это делают компьютеры!

Теперь нужно разобраться, как работает суммирование. Дополнительные задачи позволяют нам выработать новые навыки, узнать что-нибудь интересное, создать новые шаблоны.

Так что давайте рассмотрим дополнительную задачу. Мы будем использовать десятичную систему счисления.

Чтобы просуммировать $759 + 674$, я обычно складываю `digit[0]` обоих чисел, переношу единицу, затем перехожу к `digit[1]`, переношу 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 просуммированный бит 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 -му биту суммы присваивается значение 1, только если $i-1$ -е биты обоих чисел (a и b) имели значение 1. Это операция `AND` со смещением.
3. Нужно повторять эти шаги до тех пор, пока не останется переносов.

Следующий код реализует данный алгоритм.

```
1 public static int add(int a, int b) {
2     if (b == 0) return a;
3     int sum = a ^ b;           // добавляем без переноса
4     int carry = (a & b) << 1; // перенос без суммирования
5     return add(sum, carry); // рекурсия
6 }
```

Задачи, связанные с реализацией базовых операций (сложение, вычитание), достаточно популярны. Чтобы решить такую задачу, нужно разобраться с тем, как обычно реализуются операции, а потом найти путь, позволяющий написать код с учетом ограничений.

18.2. Напишите метод, тасующий карточную колоду. Колода должна быть идеально перемешана. Перестановки карт должны быть равновероятными. (Вы можете использовать идеальный генератор случайных чисел.)

Решение

Это очень популярная задача и известный алгоритм. Если вы еще не знакомы с решением, читайте дальше.

Давайте будем решать задачу «в лоб». Можно выбрать карты в произвольном порядке и поместить их в новую колоду. Фактически колода представляет собой массив, следовательно, нам нужен способ, позволяющий заблокировать отдельные элементы.

```
Исходная колода (до выбора 4): [1] [2] [3] [4] [5]
/* Выбираем случайный элемент для помещения его в начало перетасованной колоды
* Помечаем элемент в оригинальной колоде как "заблокированный", чтобы
* не выбрать его снова */
Перемешанная колода (после выбора 4): [4] [?] [?] [?] [?]
Исходная колода (после выбора 4): [1] [2] [3] [X] [5]
```

Если мы пометим элемент [4], что помешает выбрать его еще раз? Один из способов — поменять местами «мертвый» ([4]) и первый элементы колоды:

```
Исходная колода (до выбора 4): [1] [2] [3] [4] [5]
/* Выбираем случайный элемент для перемещения его в начало перетасованной колоды
* Существует элемент 1, который заменит выбранный элемент. */
Перемешанная колода (после выбора 4): [4] [?] [?] [?] [?]
Исходная колода (после выбора 4): [X] [2] [3] [1] [5]
/* Выбираем случайный элемент для перемещения его в начало
* перетасованной колоды. Есть элемент 2, который заменит только что
* выбранный элемент */
Перетасованная колода (после выбора 3): [4] [3] [?] [?] [?]
Исходная колода (после выбора 3): [X] [X] [2] [1] [5]
```

Алгоритм проще реализовать для ситуации, когда «мертвы» первые к карт, чем для ситуации, когда, например, «мертвы» третья, четвертая и девятая карты.

Оптимизировать алгоритм можно, объединив перемешанную и исходную колоды вместе.

```
Исходная колода (до выбора 4): [1] [2] [3] [4] [5]
/* Выбираем случайный элемент между 1 и 5 и меняем его местами с 1.
* В этом примере мы выбрали элемент 4.
* После этого элемент 1 — "мертв" */
Исходная колода (после выбора 4): [4] [2] [3] [1] [5]
/* Элемент 1 "мертв". Выбираем случайный элемент для замены с
* элементом 2. В этом примере пусть мы выберем элемент
* 3.*/
Исходная колода (после выбора 3): [4] [3] [2] [1] [5]
/* Повторяем. Для всех i между 0 и n-1 меняем местами случайный элемент j
* (j >= i, j < n) и элемент i. */
```

Этот алгоритм легко реализовать итеративно:

```

1 public void shuffleArray(int[] cards) {
2     int temp, index;
3     for (int i = 0; i < cards.length; i++) {
4         /* Карты с индексами от 0 до i-1 уже были выбраны
5          * (они перемещены в начало массива), поэтому сейчас мы
6          * выбираем случайную карту с индексом, больше или равным i
7          */
8         index = (int) (Math.random() * (cards.length - i)) + i;
9         temp = cards[i];
10        cards[i] = cards[index];
11        cards[index] = temp;
12    }
13 }
```

Подобный алгоритм можно придумать и самостоятельно, он достаточно часто встречается на собеседовании. Перед интервью стоит убедиться, что вы понимаете механизм его работы.

18.3. Напишите метод, генерирующий случайную последовательность m целых чисел из массива размером n . Все элементы выбираются с одинаковой вероятностью.

Решение

Первое, что приходит в голову, — выбрать случайные элементы из массива и поместить в новый массив. Но что если мы выберем один и тот же элемент дважды? В идеале, нам нужно «сократить» массив так, чтобы выкинуть выбранный элемент. Но уменьшение массива достаточно трудоемкая операция, поскольку требует смещения элементов.

Вместо того чтобы сокращать (сдвигать) массив, можно поставить элемент (поменять элементы местами) в начало массива и «запомнить», что теперь массив начинается с элемента j . Если элемент $\text{subset}[0]$ становится элементом $\text{array}[k]$, то мы должны заменить элемент $\text{array}[k]$ первым элементом в массиве. Когда мы переходим к элементу $\text{subset}[1]$, то подразумеваем, что элемент $\text{array}[0]$ «мертв», и выбираем случайный элемент y из интервала от 1 до $\text{array.size}()$. Теперь $\text{subset}[1] = \text{array}[y]$ и $\text{array}[y] = \text{array}[1]$. Элементы 0 и 1 «мертвы», а $\text{subset}[2]$ выбирается в диапазоне от $\text{array}[2]$ до $\text{array[array.size()]} = \text{array[array.size() + 1]}$ и т. д.

```

1 /* Случайное число между lower и higher включительно */
2 public static int rand(int lower, int higher) {
3     return lower + (int)(Math.random() * (higher - lower + 1));
4 }
5
6 /* Выбрать  $M$  элементов из исходного массива. Клонируем исходный
7 * массив так, чтобы не уничтожить ввод */
8 public static int[] pickMRandomly(int[] original, int m) {
9     int[] subset = new int[m];
10    int[] array = original.clone();
11    for (int j = 0; j < m; j++) {
```

продолжение ↗

```

12     int index = rand(j, array.length - 1);
13     subset[j] = array[index];
14     array[index] = array[j]; // array[j] теперь "мертв"
15   }
16   return subset;
17 }

```

- 18.4.** Напишите метод, который будет подсчитывать количество цифр «2», используемых в записи чисел от 0 до n (включительно).

Решение

Как всегда, сначала мы попробуем решить задачу «в лоб». Не забывайте, что интервьюеры хотят видеть, как вы решаете задачу.

```

1 /* Подсчитываем число '2' между 0 и n */
2 int numberOf2sInRange(int n) {
3     int count = 0;
4     for (int i = 2; i <= n; i++) { // Можем начать с 2
5         count += numberOf2s(i);
6     }
7     return count;
8 }
9
10 /* подсчитываем число '2' в одном числе */
11 int numberOf2s(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 }

```

Единственное интересное место в этом алгоритме — выделение `numberOf2s` в отдельный метод. (Это делается для чистоты кода.)

Улучшенное решение

Можно смотреть на 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 один раз из десяти.

Хотя тут стоит использовать слово «приблизительно», потому что необходимо учитывать граничные условия. Посчет количества двоек для диапазонов 1–100 и 1–37 будет различаться.

Точно количество двоек можно вычислить, рассмотрев все по отдельности разряды: $\text{digit} < 2$, $\text{digit} = 2$ и $\text{digit} > 2$.

Случай: $\text{digit} < 2$

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. Мы не будем учитывать диапазон 62000–62999. В перечисленные диапазоны попадает 6000 двоек, находящихся в 3-м разряде. Такое же количество двоек можно получить, если 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 до 10^{d+1} , а затем разделить на 10.

```
if x[d] < 2: count2sInRangeAtDigit(x, d) =
    let y = round down до ближайшего  $10^{d+1}$ 
    return y / 10
```

Случай: $\text{digit} > 2$

Давайте рассмотрим случай, когда значение d -го разряда больше, чем 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.

```
if x[d] > 2: count2sInRangeAtDigit(x, d) =
    let y = round up до ближайшего  $10^{d+1}$ 
    return y / 10
```

Случай: $\text{digit} = 2$

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).

```
if x[d] > 2: count2sInRangeAtDigit(x, d) =
    let y = округляем вниз до  $10^{d+1}$ 
    let z = правая сторона x (т.е.  $x \% 10^d$ )
    return y / 10 + z + 1
```

Теперь нам нужно пройтись по каждой цифре в числе. Реализация данного кода относительно проста:

```
1 public static 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;
```

продолжение ↗

```

9     int digit = (number / powerOf10) % 10;
10    if (digit < 2) { // если digit меньше 2
11        return roundDown / 10;
12    } else if (digit == 2) {
13        return roundDown / 10 + right + 1;
14    } else {
15        return roundUp / 10;
16    }
17 }
18 public static int count2sInRange(int number) {
19     int count = 0;
20     int len = String.valueOf(number).length();
21     for (int digit = 0; digit < len; digit++) {
22         count += count2sInRangeAtDigit(number, digit);
23     }
24     return count;
25 }
26 }
```

Данная задача требует тщательного тестирования. Убедитесь, что вы знаете все граничные случаи и проверили каждый из них.

- 18.5.** Дано: большой текстовый файл, содержащий слова. Напишите код, который позволяет найти минимальное расстояние (выражаемое количеством слов) между любыми двумя словами в файле. Достаточно ли будет $O(1)$ времени? Сколько памяти понадобится для решения?

Решение

Давайте считать, что порядок появления слов `word1` и `word2` не важен. Этот вопрос нужно согласовать с интервьюером. Если порядок слов имеет значение, нужно будет модифицировать приведенный далее код.

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 `lastPosWord1` и `lastPosWord2` соответственно. Если нам попадается `word1`, то мы сравниваем его с `lastPosWord2` и при необходимости обновляем значение `min`, а затем обновляем `lastPosWord1`. Аналогичным образом мы действуем и с `word2`. По окончании работы алгоритма в нашем распоряжении окажется правильное значение `min` (минимальное расстояние).

Приведенный далее код иллюстрирует этот алгоритм:

```

1 public int shortest(String[] words, String word1, String word2) {
2     int min = Integer.MAX_VALUE;
3     int lastPosWord1 = -1;
4     int lastPosWord2 = -1;
5     for (int i = 0; i < words.length; i++) {
6         String currentWord = words[i];
7         if (currentWord.equals(word1)) {
8             lastPosWord1 = i;
9             // Закомментируйте 3 следующие строки, если порядок слов
10            // имеет значение
```

```

10     int distance = lastPosWord1 - lastPosWord2;
11     if (lastPosWord2 >= 0 && min > distance) {
12         min = distance;
13     }
14     } else if (currentWord.equals(word2)) {
15         lastPosWord2 = i;
16         int distance = lastPosWord2 - lastPosWord1;
17         if (lastPosWord1 >= 0 && min > distance) {
18             min = distance;
19         }
20     }
21 }
22 return min;
23 }
```

Если нам придется выполнять ту же работу для других пар слов, можно создать хэштаблицу, связывающую слова с позицией в файле. Тогда решением будет минимальная (арифметическая) разница между значением из списков `listA` и `listB`.

Существует несколько способов вычислить минимальную разницу между значениями из `listA` и `listB`. Давайте рассмотрим списки:

```

listA: {1, 2, 9, 15, 25}
listB: {4, 10, 19}
```

Можно объединить списки в один отсортированный список, но связать каждое значение с исходным списком. Эта операция выполняется «обертыванием» каждого значения в класс, у которого будет две переменные экземпляра: `data` (для хранения фактического значения) и `listNumber`.

```
list: {1a, 2a, 4b, 9a, 10b, 15a, 19b, 20a}
```

Расчет минимального расстояния превращается в поиск минимального расстояния между двумя последовательными числами, у которых разные теги списка. В этом случае решением будет 1 (расстояние между `9a` и `10b`).

18.6. Опишите алгоритм для нахождения миллиона наименьших чисел в наборе из миллиарда чисел. Память компьютера позволяет хранить весь миллиард чисел.

Решение

Существует много способов решить эту задачу. Мы остановимся только на трех — сортировка, минимум кучи и ранжирование.

Решение 1. Сортировка

Можно отсортировать элементы в порядке возрастания, а затем взять первый миллион чисел. Это потребует $O(n \log(n))$ времени.

Решение 2. Минимум кучи

Чтобы решить эту задачу, можно использовать минимум кучи. Мы сначала создаем кучу для первого миллиона чисел с наибольшим элементом сверху.

Затем мы проходимся по списку. Вставляя элемент в список, удаляем наибольший элемент.

В итоге мы получим кучу, содержащую миллион наименьших чисел. Эффективность алгоритма $O(n \log(m))$, где m — количество значений, которые нужно найти.

Решение 3. Ранжирование (если изменять исходный массив)

Данный алгоритм очень популярен и позволяет найти 1-й наименьший (или наибольший) элемент в массиве.

Если элементы уникальны, поиск i-го наименьшего элемента потребует $O(n)$ времени. Основной алгоритм будет таким:

1. Выберите случайный элемент в массиве и используйте его в качестве «центра». Разбейте элементы вокруг центра, отслеживая число элементов слева.
2. Если слева находится ровно i элементов, вам нужно вернуть наибольший элемент.
3. Если слева находится больше элементов, чем i , то повторите алгоритм, но только для левой части массива.
4. Если элементов слева меньше, чем i , то повторите алгоритм справа, но ищите алгоритм с рангом $i - leftSize$.

Приведенный далее код реализует этот алгоритм.

```

1 public int partition(int[] array, int left, int right, int pivot) {
2     while (true) {
3         while (left <= right && array[left] <= pivot) {
4             left++;
5         }
6         while (left <= right && array[right] > pivot) {
7             right--;
8         }
9     }
10    if (left > right) {
11        return left - 1;
12    }
13    swap(array, left, right);
14 }
15 }
16 }
17 }
18 public int rank(int[] array, int left, int right, int rank) {
19     int pivot = array[randomIntInRange(left, right)];
20
21     /* Раздел и возврат конца левого раздела */
22     int leftEnd = partition(array, left, right, pivot);
23
24     int leftSize = leftEnd - left + 1;
25     if (leftSize == rank + 1) {
26         return max(array, left, leftEnd);
27     } else if (rank < leftSize) {
28         return rank(array, left, leftEnd, rank);

```

```

29     } else {
30         return rank(array, leftEnd + 1, right, rank - leftSize);
31     }
32 }

```

Как только найден наименьший i -й элемент, можно пройтись по массиву и найти все значения, которые меньше или равны этому элементу.

Если элементы повторяются (вряд ли они будут «уникальными»), можно слегка модифицировать алгоритма, чтобы он соответствовал этому условию. Но в этом случае невозможно будет предсказать время его выполнения.

Существует алгоритм, гарантирующий, что мы найдем наименьший i -й элемент за линейное время, независимо от «的独特性» элементов. Однако сложность этого алгоритма выходит за рамки собеседования. Если вас заинтересовала эта тема, этот алгоритм приведен в книге «CLRS' Introduction to Algorithms» (Т. Кормен, Ч. Лайзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ).

18.7. Для заданного списка слов напишите алгоритм поиска самого длинного слова, образованного другими словами, входящими в список.

Решение

Задача кажется сложной, поэтому давайте ее упростим. Допустим, что нам нужно найти самое длинное слово, составленное из двух других слов списка.

Такую задачу можно решить, пройдясь по списку, от самого длинного до самого короткого слова. Каждое слово можно разбить на возможные пары слов и проверить, есть ли обе (левая и правая) части в списке.

Псевдокод будет примерно таким:

```

1 String getLongestWord(String[] list) {
2     String[] array = list.OrderByLength();
3     /* Создаем карту для простого и быстрого поиска */
4     HashMap<String, Boolean> map = new HashMap<String, Boolean>;
5
6     for (String str : array) {
7         map.put(str, true);
8     }
9
10    for (String s : array) {
11        // Делим на каждую возможную пару
12        for (int i = 1; i < s.length(); i++) {
13            String left = s.substring(0, i);
14            String right = s.substring(i);
15            // Проверяем, есть ли обе стороны в массиве
16            if (map[left] == true && map[right] == true) {
17                return s;
18            }
19        }
20    }
21    return str;
22 }

```

Этот код работает, только когда нужно найти слово, составленное из двух других слов. Но что если количество слов будет другим?

В этом случае можно слегка модифицировать алгоритм: вместо того, чтобы искать правую часть в массиве, можно проверить, не является ли правая часть составной.

Приведенный далее код реализует этот алгоритм:

```

1 String printLongestWord(String arr[]) {
2     HashMap<String, Boolean> map = new HashMap<String, Boolean>();
3     for (String str : arr) {
4         map.put(str, true);
5     }
6     Arrays.sort(arr, new LengthComparator()); // Сортировка по длине
7     for (String s : arr) {
8         if (canBuildWord(s, true, map)) {
9             System.out.println(s);
10            return s;
11        }
12    }
13    return "";
14 }
15
16 boolean canBuildWord(String str, boolean isOriginalWord,
17 HashMap<String, Boolean> map) {
18     if (map.containsKey(str) && !isOriginalWord) {
19         return map.get(str);
20     }
21     for (int i = 1; i < str.length(); i++) {
22         String left = str.substring(0, i);
23         String right = str.substring(i);
24         if (map.containsKey(left) && map.get(left) == true &&
25             canBuildWord(right, false, map)) {
26             return true;
27         }
28     }
29     map.put(str, false);
30     return false;
31 }
```

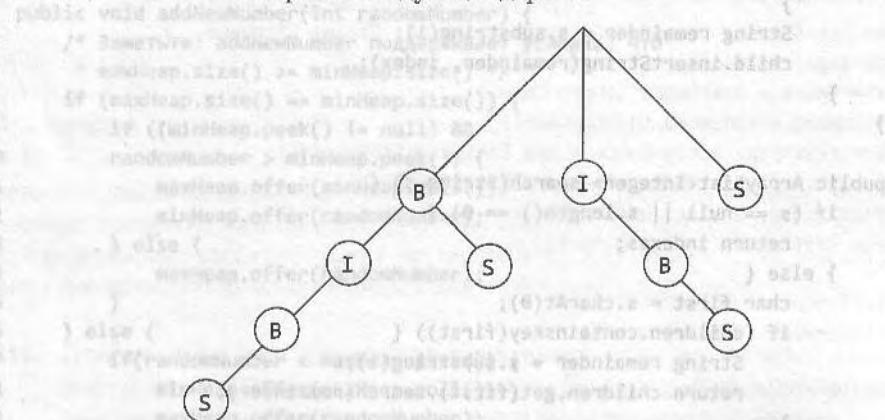
Обратите внимание, что в этом решении используется небольшая оптимизация. Мы используем динамическое программирование для кэширования результатов. Таким образом, если нам понадобится несколько раз проверить слово `testingtester`, обработка будет выполняться только один раз.

Булевый флаг `isOriginWord` управляет упомянутой оптимизацией. Метод `canBuildWord` вызывается для исходного слова и каждой подстроки. В первую очередь он проверяет кэш предварительно вычисленного результата. 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 устанавливать дополнительный флаг — `isOriginalWord`.

- 18.8. Дано: строка s и массив меньших строк T . Разработайте метод поиска s для каждой меньшей строки в T .

Решение

Прежде всего, давайте создадим суффиксное дерево для s . Например, если у нас есть слово *bibs*, мы можем построить следующее дерево:



Все, что вам нужно сделать, — произвести поиск в суффиксном дереве для каждой строки в T . Если бы T было словом, вы проверяли бы два варианта.

```

1 public class SuffixTree {
2     SuffixTreeNode root = new SuffixTreeNode();
3     public SuffixTree(String s) {
4         for (int i = 0; i < s.length(); i++) {
5             String suffix = s.substring(i);
6             root.insertString(suffix, i);
7         }
8     }
9     public ArrayList<Integer> search(String s) {
10        return root.search(s);
11    }
12 }
13 }
14
15 public class SuffixTreeNode {
16     HashMap<Character, SuffixTreeNode> children = new
17     HashMap<Character, SuffixTreeNode>();
18     char value;
19     ArrayList<Integer> indexes = new ArrayList<Integer>();
20     public SuffixTreeNode() { }
21
22     public void insertString(String s, int index) {
23         indexes.add(index);
24         if (s != null && s.length() > 0) {
25             value = s.charAt(0);
  
```

Хотя задача кажется самой простой, она является модифицируемой продолжение

```

26     SuffixTreeNode child = null;
27     if (children.containsKey(value)) {
28         child = children.get(value);
29     } else {
30         child = new SuffixTreeNode();
31         children.put(value, child);
32     }
33     String remainder = s.substring(1);
34     child.insertString(remainder, index);
35 }
36 }
37
38 public ArrayList<Integer> search(String s) {
39     if (s == null || s.length() == 0) {
40         return indexes;
41     } else {
42         char first = s.charAt(0);
43         if (children.containsKey(first)) {
44             String remainder = s.substring(1);
45             return children.get(first).search(remainder);
46         }
47     }
48     return null;
49 }
50 }

```

18.9. Сгенерированные случайным образом числа передаются методу. Напишите программу расчета среднего значения, динамически отслеживающую поступающие новые значения.

Решение

Одно из возможных решений — использовать две кучи разных приоритетов: максимальная куча (`maxHeap`) для значений выше среднего и минимальная куча (`minHeap`) для значений ниже среднего. Это позволит разделить элементы примерно поровну с двумя средними значениями — вершинами куч. Теперь найти среднее значение очень просто.

Что означает «примерно поровну»? «Примерно» означает, что при нечетном количестве чисел в одной из куч окажется лишнее число. Можно сформулировать:

- если `maxHeap.size() > minHeap.size()`, то `heap1.top()` будет средним значением;
- если `maxHeap.size() == minHeap.size()`, то средним значением будет среднее значений `maxHeap.top()` и `minHeap.top()`.

В алгоритме с балансировкой мы гарантируем, что `maxHeap` будет всегда содержать дополнительный элемент.

Алгоритм работает следующим образом. Если новое значение меньше или равно среднему, оно помещается в `maxHeap`, в противном случае оно попадает в `minHeap`. Размеры куч могут совпадать или в `maxHeap` может быть один дополнительный эле-

мент. Это требование легко выполнить, сдвигая элемент из одной кучи в другую. Среднее значение находится в вершине. Обновления занимают $O(\log(n))$ времени.

```

1 private Comparator<Integer> maxHeapComparator;
2 private Comparator<Integer> minHeapComparator;
3 private PriorityQueue<Integer> maxHeap, minHeap;
4
5 public void addNewNumber(int randomNumber) {
6     /* Заметьте: addNewNumber поддерживает условие, что
7      * maxHeap.size() >= minHeap.size() */
8     if (maxHeap.size() == minHeap.size()) {
9         if ((minHeap.peek() != null) &
10            randomNumber > minHeap.peek()) {
11             maxHeap.offer(minHeap.poll());
12             minHeap.offer(randomNumber);
13         } else {
14             maxHeap.offer(randomNumber);
15         }
16     } else {
17         if (randomNumber < maxHeap.peek()) {
18             minHeap.offer(maxHeap.poll());
19             maxHeap.offer(randomNumber);
20         }
21     } /* Конец этого блока отображения */
22     else {
23         minHeap.offer(randomNumber);
24     } /* Конец этого блока отображения */
25 }
26
27 public static double getMedian() {
28     /* maxHeap является всегда по крайней мере столь же большой,
29      * как minHeap. Если maxHeap пуста, то minHeap тоже пуста. */
30     if (maxHeap.isEmpty()) {
31         return 0;
32     }
33     if (maxHeap.size() == minHeap.size()) {
34         return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
35     } else {
36         /* Если maxHeap и minHeap разных размеров, то
37          * в maxHeap есть один дополнительный элемент.
38          * Возвращаем вершину кучи maxHeap */
39         return maxHeap.peek();
40     }
41 }
```

- 18.10.** Для двух слов одинаковой длины напишите метод, трансформирующий одно слово в другое, изменяя одну букву за один шаг. Каждое новое слово должно присутствовать в словаре.

Решение

Хотя задача кажется сложной, по сути, она является модификацией метода поиска в ширину. Каждое слово в нашем «графе» связано с другими словами из словаря,

которые отличаются от него на одну букву. Значительно интереснее сама реализация. В частности, нужно ли создавать граф?

Да, мы можем его создать, но есть более легкий путь. Используем карту поиска с возвратом. Если $v[w] = w$, вы знаете, что слово v редактировалось, чтобы получить w . Когда мы достигнем конца слова, то сможем использовать эту карту в обратном порядке, чтобы восстановить наш путь.

```

1  LinkedList<String> transform(String startWord, String stopWord,
2      Set<String> dictionary) {
3      startWord = startWord.toUpperCase();
4      stopWord = stopWord.toUpperCase();
5      Queue<String> actionQueue = new LinkedList<String>();
6      Set<String> visitedSet = new HashSet<String>();
7      Map<String, String> backtrackMap =
8          new TreeMap<String, String>();
9
10     actionQueue.add(startWord);
11     visitedSet.add(startWord);
12     String remainder;
13     while (!actionQueue.isEmpty()) {
14         String w = actionQueue.poll();
15         /* Для каждого возможного слова v из w с одной */
16         /* операцией редактирования */
17         for (String v : getOneEditWords(w)) {
18             if (v.equals(stopWord)) {
19                 // Найдено наше слово! Теперь обратно.
20                 LinkedList<String> list = new LinkedList<String>();
21                 // Добавляе v в список
22                 list.add(v);
23                 while (w != null) {
24                     list.add(0, w);
25                     w = backtrackMap.get(w);
26                 }
27             }
28             /* Если v - словарное слово */
29             if (dictionary.contains(v)) {
30                 if (!visitedSet.contains(v)) {
31                     actionQueue.add(v);
32                     visitedSet.add(v); // отмечаем посещенный
33                     backtrackMap.put(v, w);
34                 }
35             }
36         }
37     }
38     return null;
39 }
40
41 Set<String> getOneEditWords(String word) {
42     Set<String> words = new TreeSet<String>();

```

```

43     for (int i = 0; i < word.length(); i++) {
44         char[] wordArray = word.toCharArray();
45         // изменяем ту букву на что-то еще
46         for (char c = 'A'; c <= 'Z'; c++) {
47             if (c != word.charAt(i)) {
48                 wordArray[i] = c;
49                 words.add(new String(wordArray));
50             }
51         }
52     }
53     return words;
54 }

```

Пусть n — длина исходного слова, а m — число слов данного размера, находящихся в словаре. Алгоритм выполняется за $O(nm)$ времени, так как цикл while исключает из очереди m уникальных слов. Цикл for потребует $O(n)$ времени, так как он проходит строку и использует фиксированное количество замен для каждого слова.

- 18.11.** Представьте, что существует квадратная матрица, каждый пиксель которой может быть черным или белым. Разработайте алгоритм поиска максимального субквадрата, у которого все стороны черные.

Решение

Эту задачу также можно решить двумя способами: простым и сложным. Давайте рассмотрим оба решения.

«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, является ли квадрат искомым, и сообщить, если это так.

Если квадрат размером $N \times N$ не найден, можно попытаться найти следующий квадрат: $(N-1) \times (N-1)$. 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.

Наш код работает 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 стороне размером N есть (N - sz + 1) квадратов
11    * длины sz. */
12
13    return false;
}

```

предложение ↗

```

12     int count = matrix.length - squareSize + 1;
13
14     /* Перебор всех квадратов со стороной squareSize. */
15     for (int row = 0; row < count; row++) {
16         for (int col = 0; col < count; col++) {
17             if (isSquare(matrix, row, col, squareSize)) {
18                 return new Subsquare(row, col, squareSize);
19             }
20         }
21     }
22     return null;
23 }
24
25 boolean isSquare(int[][] matrix, int row, int col, int size) {
26     // Проверяем верхнюю и нижнюю стороны
27     for (int j = 0; j < size; j++) {
28         if (matrix[row][col+j] == 1) {
29             return false;
30         }
31         if (matrix[row+size-1][col+j] == 1) {
32             return false;
33         }
34     }
35
36     // Проверяем левую и правую стороны
37     for (int i = 1; i < size - 1; i++) {
38         if (matrix[row+i][col] == 1) {
39             return false;
40         }
41         if (matrix[row+i][col+size-1] == 1) {
42             return false;
43         }
44     }
45     return true;
46 }

```

Решение с предварительной обработкой: $O(N^3)$

Неторопливость «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.

`isSquare` пытается узнать, не являются ли нулевыми элементы `squareSize`, находящиеся правее (и ниже) определенных ячеек. А эту информацию можно узнать заранее.

Мы выполним проверку справа налево и снизу вверх. Для каждой ячейки нужно расчитать:

```

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

```

Посмотрите на значения для некоторой матрицы.

Обратите

(нули справа, нули слева)

мы знаем

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

Посмотрите

Исходная матрица:

W	B	W
B	B	W
B	B	W

Теперь, вместо того чтобы итерировать по $O(N)$ элементов, метод `isSquare` проверяет углы на `zerosRight` и `zerosBelow`.

Далее приведен код этого алгоритма. Обратите внимание, что `findSquare` и `findSquareWithSize` совпадают, за исключением вызова `processMatrix` и последующей работы с новым типом данных:

```

1 public class SquareCell {
2     public int zerosRight = 0;
3     public int zerosBelow = 0;
4     /* объявления, функции установки и получения значений */
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,
17 int squareSize) {
18     /* эквивалентна первому алгоритму */
19 }
20
21
22 boolean isSquare(SquareCell[][] matrix, int row, int col,
23 int size) {
24     SquareCell topLeft = matrix[row][col];
25     SquareCell topRight = matrix[row][col + size - 1];
26     SquareCell bottomRight = matrix[row + size - 1][col];
27     if (topLeft.zerosRight < size) { // Проверяем верхнюю сторону
28         return false;
29     }
30     if (topLeft.zerosBelow < size) { // Проверяем левую сторону
31         return false;
32     }
33     if (topRight.zerosBelow < size) { // Проверяем правую сторону
34         return false;
35     }
36     if (bottomRight.zerosRight < size) { // Проверяем нижнюю сторону
37         return false;
38     }
39 }
```

продолжение 

```

38     } if (count == matrix.length - 1) return true; // если в матрице одна строка
39     return true;
40 }
41
42 SquareCell[][] processSquare(int[][] matrix) {
43     SquareCell[][] processed =
44     new SquareCell[matrix.length][matrix.length];
45
46     for (int r = matrix.length - 1; r >= 0; r--) {
47         for (int c = matrix.length - 1; c >= 0; c--) {
48             int rightZeros = 0;
49             int belowZeros = 0;
50             // нужно обработать, только если ячейка черная
51             if (matrix[r][c] == 0) {
52                 rightZeros++;
53                 belowZeros++;
54                 // следующая колонка в этом ряду
55                 if (c + 1 < matrix.length) {
56                     SquareCell previous = processed[r][c + 1];
57                     rightZeros += previous.zerosRight;
58                 }
59                 if (r + 1 < matrix.length) {
60                     SquareCell previous = processed[r + 1][c];
61                     belowZeros += previous.zerosBelow;
62                 }
63             }
64             processed[r][c] = new SquareCell(rightZeros, belowZeros);
65         }
66     }
67     return processed;
68 }

```

- 18.12.** Дано: матрица размером $N \times N$, содержащая положительные и отрицательные числа. Напишите код поиска субматрицы с максимально возможной суммой.

Решение

Существует множество решений этой задачи. Мы начнем с метода грубой силы, а затем займемся оптимизацией.

Метод грубой силы: $O(N^6)$

Подобно другим задачам, связанным с поиском максимума, у этой задачи есть простое решение. Достаточно проверить все субматрицы, вычислить сумму каждой и найти самую большую.

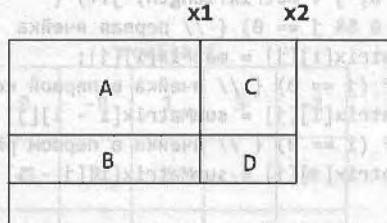
Чтобы проверить все субматрицы и избежать повторов, придется пройтись по всем упорядоченным парам строк и затем по всем упорядоченным парам столбцов.

Это решение потребует $O(N^6)$ времени, так как необходимо проверить $O(N^4)$ матриц, а проверка одной матрицы занимает $O(N^2)$ времени.

Решение динамического программирования: $O(N^4)$

Обратите внимание, что предыдущее решение работает медленно из-за рассчета суммы элементов матрицы — $O(N^2)$ — очень медленная операция. Можно ли сократить это время? Да! Мы можем уменьшить время `computeSum` до $O(1)$.

Посмотрите на следующий прямоугольник:



Предположим, что нам 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 `Val*` начинается в исходной точке и заканчивается в нижнем правом углу подпрямоугольника.

Про эти значения мы знаем следующее:

$$\text{area}(D) = \text{ValD} - \text{area}(A \cup C) + \text{area}(A \cup B) - \text{area}(A)$$

Или:

$$\text{area}(D) = \text{ValD} - \text{ValB} - \text{ValC} + \text{ValA}$$

Данная информация позволит эффективно рассчитать эти значения для всех точек матрицы:

$$\text{Val}(x, y) = \text{Val}(x - 1, y) + \text{Val}(y - 1, x) - \text{Val}(x - 1, y - 1)$$

Можно заранее рассчитать подобные значения и затем найти максимальную субматрицу.

Следующий код реализует данный алгоритм.

```
1 int getMaxMatrix(int[][] original) {
2     int maxArea = Integer.MIN_VALUE; // Важно! Max может быть < 0
3     int rowCount = original.length;
4     int columnCount = original[0].length;
5     int[][] matrix = precomputeMatrix(original);
6     for (int row1 = 0; row1 < rowCount; row1++) {
7         for (int row2 = row1; row2 < rowCount; row2++) {
8             for (int col1 = 0; col1 < columnCount; col1++) {
9                 for (int col2 = col1; col2 < columnCount; col2++) {
10                     maxArea = Math.max(maxArea, computeSum(matrix,
11                         row1, row2, col1, col2));
12                 }
13             }
14         }
15     }
```

продолжение ➔

```

16     return maxArea;
17 }
18
19 int[][] precomputeMatrix(int[][] matrix) {
20     int[][] sumMatrix = new int[matrix.length][matrix[0].length];
21     for (int i = 0; i < matrix.length; i++) {
22         for (int j = 0; j < matrix.length; j++) {
23             if (i == 0 && j == 0) { // первая ячейка
24                 sumMatrix[i][j] = matrix[i][j];
25             } else if (j == 0) { // ячейка в первой колонке
26                 sumMatrix[i][j] = sumMatrix[i - 1][j] + matrix[i][j];
27             } else if (i == 0) { // ячейка в первом ряду
28                 sumMatrix[i][j] = sumMatrix[i][j - 1] + matrix[i][j];
29             } else {
30                 sumMatrix[i][j] = sumMatrix[i - 1][j] +
31                     sumMatrix[i][j - 1] - sumMatrix[i - 1][j - 1] +
32                     matrix[i][j];
33             }
34         }
35     }
36     return sumMatrix;
37 }
38
39 int computeSum(int[][] sumMatrix, int i1, int i2, int j1, int j2) {
40     if (i1 == 0 && j1 == 0) { // начинаем с ряда 0, колонки 0
41         return sumMatrix[i2][j2];
42     } else if (i1 == 0) { // начинаем с ряда 0
43         return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1];
44     } else if (j1 == 0) { // начинаем с колонки 0
45         return sumMatrix[i2][j2] - sumMatrix[i1 - 1][j2];
46     } else {
47         return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1] -
48             sumMatrix[i1 - 1][j2] + sumMatrix[i1 - 1][j1 - 1];
49     }
50 }

```

Оптимизированное решение: $O(N^3)$

Невероятно, но существует еще более оптимальное решение. Если у нас есть R строк и C столбцов, то задачу можно решить за $O(R^2C)$ времени.

Вспомните решение задачи про поиск максимального субмассива для массива целых чисел (`integer`) найдите субмассив с максимальной суммой. Такой максимальный субмассив можно найти за $O(N)$ времени. Давайте используем это решение для нашей задачи.

Каждую субматрицу можно представить в виде последовательности строк и последовательности столбцов. Можно пройтись по строкам и найти столбцы, дающие максимальную сумму. Код будет таким:

```

1 maxSum = 0
2 foreach rowStart in rows
3     foreach rowEnd in rows
4     /* У нас есть количество возможных субматриц с границами

```

```

5 * rowStart и rowEnd
6 * Найдите границы colStart и colEnd, дающие
7 * максимальную сумму. */
8 maxSum = max(runningMaxSum, maxSum)
9 return maxSum

```

Теперь остается вопрос: как найти «лучшие» colStart и colEnd?

Рассмотрим субматрицу:

rowStart				
9	-8	1	3	-2
-3	7	6	-2	4
6	-4	-4	8	-7
12	-5	3	9	-5
rowEnd				

Нам необходимо найти colStart и colEnd, которые дают нам максимально возможную сумму всех субматриц с rowStart сверху и rowEnd снизу. Можно вычислить сумму каждого столбца и использовать функцию maximumSubArray, которая обсуждалась в начале решения этой задачи.

В предыдущем примере максимальный субмассив охватывал пространство с первой по четвертую колонку. Это означает, что максимальная субматрица должна простираться от (rowStart, первый столбец) до (rowEnd, четвертый столбец).

Теперь мы можем записать следующий псевдокод:

```

1 maxSum = 0
2 foreach rowStart in rows
3   foreach rowEnd in rows
4     foreach col in columns
5       partialSum[col] = sum of matrix[rowStart, col] through
6       matrix[rowEnd, col]
7       runningMaxSum = maxSubArray(partialSum)
8       maxSum = max(runningMaxSum, maxSum)
9   return maxSum

```

Вычисление суммы в строках 5 и 6 занимает $R \times C$ времени (так как требует итерации от rowStart до rowEnd), что дает общее время выполнения $O(R^3C)$.

В строках 5 и 6 мы добавляли $a[0] \dots a[i]$ с нуля, даже если на предыдущей итерации внешнего цикла добавились $a[0] \dots a[i-1]$. Давайте избавимся от двойной работы.

```

1 maxSum = 0
2 foreach rowStart in rows
3   clear array 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

```

Полная версия кода выглядит так:

```

1 public void clearArray(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3         array[i] = 0;
4     }
5 }
6
7 public static int maxSubMatrix(int[][] matrix) {
8     int rowCount = matrix.length;
9     int colCount = matrix[0].length;
10
11    int[] partialSum = new int[colCount];
12    int maxSum = 0; // Макс. сумма = 0 (матрица пуста)
13
14    for (int rowStart = 0; rowStart < rowCount; rowStart++) {
15        clearArray(partialSum);
16
17        for (int rowEnd = rowStart; rowEnd < rowCount; rowEnd++) {
18            for (int i = 0; i < colCount; i++) {
19                partialSum[i] += matrix[rowEnd][i];
20            }
21
22            int tempMaxSum = maxSubArray(partialSum, colCount);
23
24            /* Если вы хотите отслеживать координаты, добавьте некие "глобальные" переменные
25             * код здесь, чтобы сделать это. */
26            maxSum = Math.max(maxSum, tempMaxSum);
27        }
28    }
29    return maxSum;
30 }
31
32 public static int maxSubArray(int array[], int N) {
33     int maxSum = 0;
34     int runningSum = 0;
35
36     for (int i = 0; i < N; i++) {
37         runningSum += array[i];
38         maxSum = Math.max(maxSum, runningSum);
39
40         /* Если runningSum < 0, нет смысла продолжать ряд
41          * Сброс. */
42         if (runningSum < 0) {
43             runningSum = 0;
44         }
45     }
46     return maxSum;
47 }
```

Это чрезвычайно сложная задача. Вряд ли вы сможете решить подобную задачу на собеседовании без подсказки интервьюера.

- 18.13.** Дан список из миллиона слов. Разработайте алгоритм, создающий максимальный 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

Большинство задач, использующих словарь, требуют некоторой предварительной обработки.

Как можно провести предварительную обработку?

Если мы собираемся создать квадрат из слов, то длина всех строк и высота всех столбцов должны быть одинаковыми. Давайте 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 $(\text{length}(\text{longestWord}))^2$.

```
1 int maxRectangle = longestWord * longestWord;
2 for z = maxRectangle to 1 {
3     for each pair of numbers (i, j) where i*j = z {
4         /* пытаемся создать прямоугольник, возвращаемся, если успешно */
5     }
6 }
```

Мы проходим по прямоугольникам от самого большого до самого маленького, таким образом первый найденный прямоугольник будет самым большим.

Теперь самая сложная часть — `makeRectangle(int l, int h)`. Этот метод пытается создать прямоугольник из слов размером $l \times h$.

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, что мы пытаемся создать прямоугольник размером 6×5 и первыми парами строк будут:

```
there
queen
pizza
....
```

В этой точке мы уже знаем, что первый столбец начинается с `tqp`. Мы знаем (или должны знать), что ни одно из слов в словаре не начинается с `tqp`. Зачем мы продолжили создавать прямоугольник, если знали, что у нас не получится создать допустимый прямоугольник?

Значит, нужна оптимизация. Можно создать выборку, позволяющую упростить поиск, если будем анализировать подстроки как префиксы слов в словаре. При построчном формировании прямоугольника можно ввести проверку, являются ли столбцы допустимыми префиксами. Если нет, мы сразу прекращаем работу с этим прямоугольником.

Приведенный далее код реализует этот алгоритм. Это длинный и сложный алгоритм, поэтому мы будем анализировать его по частям.

Прежде всего, нам необходима предварительная обработка, позволяющая сгруппировать слова по длине. Мы создаем массив выборок (по одной на каждую длину слова), но пока не будем их использовать.

```
1 WordGroup[] groupList = WordGroup.createWordGroups(list);
2 int maxWordLength = groupList.length;
3 Trie trieList[] = new Trie[maxWordLength];
```

Метод `maxRectangle` — главная часть нашего кода. Он начинает работу с самого большого возможного прямоугольника (`maxWordLength2`) и пытается построить прямоугольник этого размера. Если это невозможно, он пытается создать прямоугольник меньшего размера. Первый прямоугольник, который удастся построить, будет самым большим.

```
1 Rectangle maxRectangle() {
2     int maxSize = maxWordLength * maxWordLength;
3     for (int z = maxSize; z > 0; z--) { // начинаем с наибольшей области
4         for (int i = 1; i <= maxWordLength; i++) {
5             if (z % i == 0) {
6                 int j = z / i;
7                 if (j <= maxWordLength) {
8                     /* Создаем прямоугольник длиной i и высотой j.
9                      * Заметьте, что i * j = z. */
10                    Rectangle rectangle = makeRectangle(i, j);
11                    if (rectangle != null) {
12                        return rectangle;
13                    }
14                }
15            }
16        }
17    }
18    return null;
19 }
```

`maxRectangle` вызывает метод `makeRectangle` и пытается построить прямоугольник указанных размеров.

```
1 Rectangle makeRectangle(int length, int height) {
2     if (groupList[length - 1] == null ||
3         groupList[height - 1] == null) {
4         return null;
5     }
6
7     /* Создает выборку для длины слова, если мы ее еще не создали */
8     if (trieList[height - 1] == null) {
9         LinkedList<String> words = groupList[height - 1].getWords();
10        trieList[height - 1] = new Trie(words);
11    }
12
13    return makePartialRectangle(length, height,
14        new Rectangle(length));
15 }
```

Метод `makePartialRectangle` — наш основной метод, производящий всю работу. Ему передаются окончательные значения длины и высоты, а также частично сформированный прямоугольник. Если нам известно окончательное значение высоты

прямоугольника, то мы должны проверить, что колонки содержат допустимые слова, и выйти.

В противоположном случае мы проверяем, сформулированы ли столбцы из допустимых префиксов. Если нет, работа останавливается, поскольку нет смысла продолжать строить этот прямоугольник.

Если все нормально и все колонки содержат правильные префиксы, мы перебираем все слова нужной длины, добавляем каждое из них к прямоугольнику и пытаемся построить новый прямоугольник (текущий прямоугольник с новым добавленным словом).

```

1 Rectangle makePartialRectangle(int l, int h, Rectangle rectangle) {
2     if (rectangle.height == h) { // Проверяем, полный ли прямоугольник
3         if (rectangle.isComplete(l, h, groupList[h - 1])) {
4             return rectangle;
5         } else {
6             return null;
7         }
8     }
9
10    /* Сравниваем колонки с выборкой, чтобы увидеть */
11    /* потенциально допустимый прямоугольник */
12    if (!rectangle.isPartialOK(l, trieList[h - 1])) {
13        return null;
14    }
15    /* Проходимся по всем словам нужной длины. Добавляем каждое в */
16    /* текущий частичный прямоугольник и пытаемся построить прямоугольник */
17    /* рекурсивно. */
18    for (int i = 0; i < groupList[l-1].length(); i++) {
19        /* Создаем новый прямоугольник, добавляя новое слово в текущий */
20        Rectangle orgPlus =
21            rectangle.append(groupList[l-1].getWord(i));
22
23        /* Пытаемся построить прямоугольник с этим новым, */
24        /* частичным прямоугольником */
25        Rectangle rect = makePartialRectangle(l, h, orgPlus);
26        if (rect != null) {
27            return rect;
28        }
29    }
30    return null;
31 }
```

Класс `Rectangle` представляет собой частично или полностью сформированный прямоугольник из слов. Метод `isPartialOK` вызывается для проверки допустимости прямоугольника. Метод `isComplete` выполняет аналогичную функцию, но дополнительно проверяет, чтобы колонки содержали полное слово.

```

1 public class Rectangle {
2     public int height, length;
3     public char [][] matrix;
4
5     /* Создаем "пустой" прямоугольник. Длина - фиксированная, но высота */
6     /* может изменяться при добавлении слов */
7 }
```

```

7     public Rectangle(int l) {
8         height = 0;
9         length = l;
10    }
11
12   /* Создаем прямоугольный массив слов
13  * определенной длины и высоты
14  * (Предполагается, что длина и высота определены
15  * как аргументы и не противоречат
16  * размерам массива) */
17  public Rectangle(int length, int height, char[][] letters) {
18      this.height = letters.length;
19      this.length = letters[0].length;
20      matrix = letters;
21  }
22
23  public char getLetter (int i, int j) { return matrix[i][j]; }
24  public String getColumn(int i) { ... }
25
26  /* Проверяем, все ли колонки допустимы. Все строки будут
27  * допустимы, так как они были добавлены непосредственно из словаря */
28  public boolean isComplete(int l, int h, WordGroup groupList) {
29      if (height == h) {
30          /* Проверяем, является ли каждая колонка словарным словом*/
31          for (int i = 0; i < l; i++) {
32              String col = getColumn(i);
33              if (!groupList.containsWord(col)) {
34                  return false;
35              }
36          }
37          return true;
38      }
39      return false;
40  }
41  public void setLength(int l) {
42  public boolean isPartialOK(int l, Trie trie) {
43      if (height == 0) return true;
44      for (int i = 0; i < l; i++) {
45          String col = getColumn(i);
46          if (!trie.contains(col)) {
47              return false;
48          }
49      }
50      return true;
51  }
52
53  /* Создаем новый Rectangle: берем строки текущего
54  * прямоугольника и добавляем s. */
55  public Rectangle append(String s) { ... }
56 }

```

Класс `WordGroup` — контейнер, содержащий слова определенной длины. Для упрощения поиска мы будем хранить слова в хэш-таблице так же, как в `ArrayList`.

Списки в `WordGroup` создаются с помощью статического метода `createWordGroups`.

```

1  public class WordGroup {
2      private Hashtable<String, Boolean> lookup = new Hashtable<String, Boolean>();
3      private ArrayList<String> group = new ArrayList<String>();
4
5
6      public boolean containsWord(String s) {
7          return lookup.containsKey(s));
8      }
9
10     public void addWord (String s) {
11         group.add(s);
12         lookup.put(s, true);
13     }
14
15     public int length() { return group.size(); }
16     public String getWord(int i) { return group.get(i); }
17     public ArrayList<String> getWords() { return group; }
18
19     public static WordGroup[] createWordGroups(String[] list) {
20         WordGroup[] groupList;
21         int maxWordLength = 0;
22         /* Находим длину самого длинного слова */
23         for (int i = 0; i < list.length; i++) {
24             if (list[i].length() > maxWordLength) {
25                 maxWordLength = list[i].length();
26             }
27         }
28
29         /* Группируем слова в словаре в списки одинаковой длины
30         * length.groupList[i] будет содержать список слов
31         * длиной (i+1). */
32         groupList = new WordGroup[maxWordLength];
33         for (int i = 0; i < list.length; i++) {
34             /* Мы делаем wordLength - 1 вместо просто wordLength, так как
35             * мы используем wordLength и нет слов длиной 0 */
36             int wordLength = list[i].length() - 1;
37             if (groupList[wordLength] == null) {
38                 groupList[wordLength] = new WordGroup();
39             }
40             groupList[wordLength].addWord(list[i]);
41         }
42         return groupList;
43     }
44 }
```

Полный код для этой задачи, включая коды методов `Trie` и `TrieNode`, вы можете скачать с сайта автора книги. Не забудьте, что в подобных сложных задачах лучше использовать псевдокод. На написание полного кода вам просто не хватит времени.

Благодарности

Все в этой жизни делается «в команде», и эта книга — не исключение. Многие люди помогали мне в работе над ней, и я хотела бы выразить им свою искреннюю благодарность.

Во-первых, я хочу поблагодарить Джона — моего мужа, который поддерживал меня во всем. Без него эта книга просто не была бы написана.

Во-вторых, я благодарна моей маме, которая научила меня, что код — это важно, но еще более важно — уметь излагать свои мысли на родном языке. Она — отличный инженер, предприниматель, а самое главное — замечательная мама.

В-третьих, я хочу поблагодарить всех моих друзей за поддержку.

Наконец, что не менее важно, хочу выразить благодарность всем читателям за отзывы и предложения. Особенно я хотела бы поблагодарить Винита Шаха и Пренея Варму, написавших замечательный обзор вопросов, использованных в книге. Вы пошли намного дальше, чем того требовали ваши служебные обязанности. Вашим коллегам и руководителям невероятно повезло.

Еще раз огромное спасибо!

```

27  * запускаем, как мы делаем проверку в main() в главной функции
28 public boolean isComplete(int[] list, int n, Map<Group, groupList> {
29     if (height >= 255) return true;
30     /* Проверяем, является ли каждая колонка сложной */
31     for (int i = 0; i < n; i++) {
32         String col = getRow(i); // строка отсортированная
33         if (!groupList.containsKey(height + i)) continue;
34         return false;
35     }
36     /* Проверка, есть ли в таблице хотя бы одна строка,
37      в которой все ячейки в колонках с одинаковым номером
38      являются «стекающими» */
39     return false;
40 }
41 public boolean isPartialOK(int[] list, int height) {
42     Map<Group, groupList> map = new HashMap<Group, groupList>();
43     if (height == 0) return true;
44     for (int i = 0; i < height; i++) {
45         String row = getRow(i);
46         if (!map.containsKey(row)) map.put(row, new ArrayList<Group>());
47         if (!map.get(row).contains(list[i])) map.get(row).add(list[i]);
48     }
49     for (String row : map.keySet()) {
50         if (isPartialOK(map.get(row), height)) return true;
51     }
52     /* Создаем новый Rectangle: берем строки текущего
53     * прямоугольника и добавляем в */
54 }
```

Без этого кода не получится пройти тесты на языке Java, края ячеек которых предстоит поместить в соответствующие ячейки в списке. А это, в свою очередь, означает, что это не просто логика, это логика, которая должна быть встроена в классы, описанные в главе 11. Доказательство

Об авторе

Гэйл Лакман Макдаулл занималась разработкой программного обеспечения в Microsoft, Apple и Google.

Последние три года она разрабатывает программное обеспечение для Google и по совместительству является одним из ведущих интервьюеров этой компании, принимая активное участие в подборе кадров. Она провела более чем 150 собеседований с кандидатами как в США, так и за границей, оценила более 1000 пакетов документов, присыпаемых соискателями, и в сотни раз больше резюме.

Ее приглашали на работу ведущие ИТ-компании — Microsoft, Google, Amazon, IBM и Apple.

В 2005 году Макдаулл основала CareerCup.com, чтобы поделиться накопленным опытом с теми, кто ищет работу в области ИТ-бизнеса.

Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/efpartners

и получите бесплатный тест

«Почему я могу быть успешным партнером?»
Выбирайте книги на сайте www.piter.com, размещайте информацию о книге на своем сайте, включайте ссылку на книгу в другие и добавляйте в тексте ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! Для получения дополнительной информации досмотрите видео «Как стать партнером» на канале YouTube Партнера.

Бонусом Вашего успеха

будет получение

привилегий

партнера

для продажи

книг

на вашем

сайте

С этого момента получайте 10% от стоимости каждой продажи, которую совершил покупатель приобретя книгу в интернет-магазине «Питер» позже либо с Вашим партнерским номером. А если покупатель приобретает не только эту книгу, но и другие издания, Вы получаете дополнительное 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издавательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или WebMoney.

Лимит перевода — 500 000 руб. (или 10 000 долларов США). Минимальная сумма перевода — 100 рублей.
Пример: «Питер-Интернет-Магазин» — это личный кабинет пользователя — есть такое же имя в системе Яндекс.Деньги и WebMoney. Введите его в поле «Номер счета» в разделе «Перевод на кошелек» в меню «Переводы».

<http://www.piter.com/efpartners> — партнерская ссылка, где «000» — это ваш личный партнерский номер.

Подробнее в Партнерской программе
ИД «Питер» читайте на сайте
www.piter.com.

Издательский дом
ПИТЕР®
www.piter.com